View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Universidad de Zaragoza

PROYECTO FIN DE CARRERA

PARALELIZACIÓN DEL ALGORITMO DE BÚSQUEDA DE UN RECONOCEDOR AUTOMÁTICO DE VOZ

Autor
Juan Vallés Martín

Director
Antonio Miguel Artiaga

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2014

Juan Vallés Martín: *Paralelización del algoritmo de búsqueda de un reconocedor automático de voz*, 2014

RESUMEN

Paralelización del algoritmo de búsqueda de un reconocedor automático de voz

Durante años, la velocidad de los procesadores ha aumentado debido al aumento de transistores en los circuitos integrados. Estas mejoras en la eficiencia no requerían cambios en el software: el mismo programa era más rápido en un ordenador con una frecuencia de reloj más alta. Sin embargo, la posibilidad de seguir mejorando la capacidad de los sistemas actuales puede ser acelerada a un ritmo mucho mayor si se consigue paralelizar el problema y tratarlo mediante arquitecturas de hardware paralelo disponibles, como procesadores multinúcleo, clústers o GPUs (*Graphics Processing Units*).

El proyecto plantea el estudio de la viabilidad de un reconocedor de voz con funciones en paralelo mediante el desarrollo de un prototipo. Los objetivos principales son la paralelización de la búsqueda de la secuencia de estados (sonidos) más probable y el cálculo de las verosimilitudes de los datos de entrada (observaciones), explorando las posibilidades que este paralelismo ofrece y viendo el rendimiento que con él puede llegarse a obtener. El desarrollo se lleva a cabo en el lenguaje de programación C, mientras que las funciones paralelizadas se implementan en GPUs utilizando CUDA, un modelo de programación adaptado a esta arquitectura, y su extensión para C.

En cada instante del proceso de reconocimiento hay un número determinado de tokens activos con una probabilidad y una secuencia de estados asociadas y que representan las hipótesis más probables hasta el momento. Cuando hay nuevos datos de entrada, estos tokens se propagan hacia los siguientes estados, cambiando su peso dependiendo de las probabilidades de transición entre estados y de las probabilidades de observación (cómo los datos se ajustan al sonido correspondiente a cada estado). Los tokens que acaban en el mismo estado que otro con mayor peso y los que no superan cierta probabilidad son desechados. El proceso acaba mostrando la secuencia asociada al token de mayor peso en el instante final. Hay partes de este proceso que son expresables como productos matriciales o vectoriales y que por tanto son fácilmente paralelizables.

Cada estado lleva asociada una mezcla de Gaussianas de las mismas dimensiones que las de los datos de entrada. La parte más costosa del cálculo de las probabilidades de observación es una distancia entre vectores de muchas dimensiones. Desarrollándola como un polinomio de segundo grado y apilando los coeficientes de todos los polinomios en una matriz podemos convertir este cálculo en un producto matricial, susceptible también de ser paralelizado.

ÍNDICE GENERAL

1 INTRODUCCIÓN 1
1.1 Antecedentes y motivación 1
1.2 Descripción y objetivos del proyecto 2
1.3 Organización de la memoria 3

2 EL PROCESO DE RECONOCIMIENTO 5
2.1 Fundamento teórico 5
2.2 Modelos Ocultos de Markov 6
2.3 Algoritmo de Viterbi 9
2.4 Búsqueda con tokens 11

3 IMPLEMENTACIÓN 15
3.1 Introducción 15
3.2 Consideraciones previas a la implementación 16
3.3 Propagación 19
3.3.1 Cálculo de tokens en el siguiente frame 19
3.3.2 Purga 20
3.3.3 Búsqueda del máximo por reducción 21
3.3.4 Actualización de índices y número de tokens activos 23
3.4 Cálculo de las probabilidades de observación 23
3.4.1 Inicialización de `gMask` y `fetch` 24
3.4.2 Multiplicación con máscara 24
3.4.3 Suma y normalización 25
3.4.4 Actualización de las probabilidades acumuladas 26
3.5 Recuperación de resultados 26
3.5.1 Actualización del buffer Φ 26
3.5.2 Algoritmo de bactracking 28

4 RESULTADOS 31
4.1 Estudio de tiempos 31
4.1.1 Comparación del rendimiento con otros reconocedores 32
4.1.2 Distribución de tiempos 33
4.1.3 Rendimiento del cálculo de probabilidades de observación 35

5 CONCLUSIONES 37
5.1 Resumen del proyecto y análisis de objetivos 37
5.2 Desarrollo en el futuro 38

A CONCEPTOS BÁSICOS DE CUDA 41

A.1 Historia de la programación en paralelo 41
A.2 Introducción a CUDA C 43
A.3 Ejemplo: suma de vectores 44

B CÁLCULO DE LAS PROBABILIDADES DE OBSERVACIÓN 47
B.1 Introducción 47
B.2 Expresión del cálculo como producto matricial 47

C ALGORITMO DE RECONOCIMIENTO 49
C.1 Consideraciones previas 49
C.2 Algoritmo 50
C.2.1 Inicialización 50
C.2.2 Bucle de reconocimiento 51
C.2.3 Backtracking 52

D ESTRUCTURAS DE DATOS 55

BIBLIOGRAFÍA 59

1 INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

La interacción entre personas y máquinas es más frecuente y diversa a medida que avanza la tecnología. Hoy podemos pedirle a un dispositivo que nos indique la ruta hacia nuestro destino o que reconozca una canción por nosotros. Dado que el habla es, en muchas situaciones, la forma de interacción más natural para el ser humano, es lógico que haya un interés especial en el desarrollo de sistemas capaces de reconocer y entender la voz humana.

El Reconocimiento Automático del Habla (RAH) es el proceso de clasificación de secuencias de patrones extraídas de una señal de audio que contiene voz humana, de forma que el mensaje contenido en ella es reconocido. Entre sus aplicaciones pueden encontrarse sistemas de dictado de palabras o documentos, traducción entre lenguajes, sistemas de control por voz o subtitulado automático de documentos audiovisuales.

Aunque la investigación en este campo comenzó hace décadas, y pese a los avances conseguidos en los últimos años, todavía son necesarias mejoras en la robustez y la velocidad de los sistemas de reconocimiento para poder hablar de un reconocedor de altas prestaciones, especialmente si se consideran aplicaciones de tiempo real. El proyecto plantea la programación de funciones en paralelo como medio para acelerar el software de reconocimiento del habla.

Hasta hace algunos años la velocidad de los procesadores aumentaba principalmente debido al aumento del número de transistores en los circuitos integrados (aproximadamente el doble cada dos años), siguiendo la ley de Moore. De esta forma, el mismo programa era más rápido en un ordenador de prestaciones más altas sin requerir cambios en el software. Sin embargo, aunque el número de transistores continúa aumentando, la velocidad de reloj ha dejado de seguir esa tendencia, y recientemente han aparecido nuevas arquitecturas y modelos de programación que permiten aumentar la velocidad de los sistemas de una forma alternativa y, en ocasiones, a un ritmo mucho mayor.

La programación en paralelo, y en particular la programación de GPUs (Graphic Processing Units), es un modelo que ha ganado popularidad en los últimos años. Dado que la mayoría de operaciones realizadas sobre un píxel en una imagen no dependen del resultado de dicha operación en otros píxeles, las tarjetas gráficas se compo-

nen de varios procesadores (más simples que una CPU) capaces de realizar la misma tarea en paralelo, de forma que varios píxeles son tratados a la vez. Esta idea se ha aprovechado en aplicaciones tradicionalmente ejecutadas por CPUs consiguiendo mejoras en los tiempos de ejecución. En particular, aquellas funciones expresables como sumas o productos matriciales dan buenos resultados al programarse en paralelo. Actualmente el framework que predomina en la programación de GPUs es *CUDA*, perteneciente a nVidia[1],, el cual proporciona una serie de herramientas de desarrollo para acceder a los sets de instrucciones de sus tarjetas gráficas.

1.2 DESCRIPCIÓN Y OBJETIVOS DEL PROYECTO

Como se mencionaba en el apartado anterior, el objetivo fundamental del proyecto es el estudio de la viabilidad de un reconocedor de voz con los distintos pasos del proceso de reconocimiento implementados como funciones en paralelo programadas en CUDA, observando las mejoras en los tiempos de ejecución que con éstas se pueden conseguir. Para ello, es necesario crear un prototipo programado en lenguaje C, capaz de reconocer secuencias de palabras a partir de ejemplos y modelos estadísticos del laboratorio de Tecnologías del Habla[2] del Grupo de Tecnologías de las Comunicaciones de la Universidad de Zaragoza.

El primer objetivo del proyecto es, por tanto, expresar el algoritmo de reconocimiento (tradicionalmente implementado como una serie de bucles anidados) como una sucesión de operaciones matriciales y vectoriales, para así facilitar el desarrollo posterior de funciones en paralelo. Las estructuras de datos resultantes son matrices y vectores de gran tamaño pero con solamente unos pocos elementos distintos de cero, por lo que es necesario trabajar con estructuras de tipo *sparse*, las cuales guardan únicamente los índices y los valores de los elementos activos en un vector o en una matriz. Existen librerías que permiten trabajar con este tipo de estructuras en CUDA y realizar operaciones matriciales simples como sumas o multiplicaciones. Las funciones disponibles en esta librería son a priori más rápidas que cualquier versión "hecha a mano"de las mismas, aunque son funciones muy generales y optimizadas para la resolución de sistemas de ecuaciones. Por otra parte, no permiten trabajar en escala logarítmica, lo cual es un problema, ya que en el reconocimiento de voz los cálculos en escala lineal pueden salirse fácilmente de rango. El siguiente objetivo es, por tanto, la comparación de una solución basada en librerías con otra basada en funciones y estructuras hechas a medida para el problema.

---

1 http://www.nvidia.com/object/cuda_home_new.html
2 http://vivolab.es/

Para facilitar el proceso de depurado también es conveniente tener una versión en Matlab del prototipo (sin funciones en paralelo, aunque con su equivalente matricial), ya que es más sencillo de programar aunque mucho menos eficiente. Asimismo, hace falta un programa, que también se implementa en Matlab, que lea los datos (los modelos acústicos y de lenguaje y las distintas secuencias de patrones) en el formato que utiliza HTK [6], un software de manejo de modelos de Markov utilizado habitualmente en RAH, y las transforme al formato adecuado para la versión en C del reconocedor.

Entre las funciones programadas en paralelo se distinguen dos partes: el algoritmo de Viterbi y el cálculo de las probabilidades de observación. En el proceso de reconocimiento (explicado con detalle en el capítulo 2), la producción del habla se modela mediante Modelos Ocultos de Markov (HMM), y la búsqueda de la secuencia de palabras que mejor explica los datos observados se realiza mediante el algoritmo de Viterbi, aunque debido al tamaño de la red de estados (o sonidos posibles) que se maneja en un caso normal de reconocimiento es necesario incluir en él ciertas modificaciones. Hay pasos en este algoritmo claramente paralelizables, como la propagación de un estado hacia sus posibles estados destino. Otros, como la búsqueda de la hipótesis más probable entre las que consideramos en un momento dado, tienen una componente secuencial que dificulta su paralelización. Sin embargo, ya que la transferencia de datos entre CPU y GPU es costosa en tiempo, todos los pasos del algoritmo se intentarán programar con funciones en paralelo para mantener el tiempo dedicado a transferir datos en el mínimo necesario.

La paralelización del cálculo de las probabilidades de observación es uno de los últimos objetivos que se incluyeron en el proyecto. Este cálculo se realiza en cada iteración de la fase de búsqueda del algoritmo de Viterbi, pero inicialmente se consideró calcularlo en CPU. Sin embargo, la carga computacional de esta operación hace que las mejoras en su eficiencia tengan un impacto importante en el rendimiento total del programa. La tarea de esta función consiste en el cálculo de la probabilidad de que unos datos de entrada correspondan a cada uno de los estados que consideramos posibles en un determinado momento. La parte más costosa de este cálculo es la distancia entre dos vectores de muchas dimensiones, la cual puede desarrollarse como un polinomio de segundo grado. Escribiendo los coeficientes de forma matricial, el cálculo puede realizarse como un producto, que como ya se ha mencionado es fácilmente paralelizable.

1.3 ORGANIZACIÓN DE LA MEMORIA

Aparte del presente capítulo, que sirve como introducción y resumen del proyecto, la memoria se organiza en las siguientes secciones:

- En el Capítulo 2 se desarrollan los fundamentos teóricos de los sistemas de RAH como los HMMs, o la búsqueda de Viterbi.
- El Capítulo 3 explica los detalles de la implementación del prototipo de reconocedor basado en funciones en paralelo.
- En el Capítulo 4 se presentan y analizan los resultados del trabajo desarrollado, se muestran los distintos tiempos de ejecución del programa y se comparan con los de otros reconocedores.
- El Capítulo 5, finalmente, presenta las conclusiones del proyecto, repasa los objetivos iniciales de éste y su desarrollo y ofrece una serie de líneas de trabajo futuras.
- En el Apéndice A puede encontrarse una breve introducción a la programación en paralelo basada en GPUs, con un ejemplo de función escrita en CUDA C.
- En el Apéndice B se desarrolla el cálculo de las probabilidades de observación como un producto matricial.
- El Apéndice C explica detalladamente el algoritmo de reconocimiento basado en matrices implementado durante el proyecto.
- En el Apéndice D aparecen detalladas las distintas estructuras de datos creadas para el desarrollo del prototipo.

2 EL PROCESO DE RECONOCIMIENTO

2.1 FUNDAMENTO TEÓRICO

Un Reconocedor Automático del Habla (RAH) es un sistema que intenta extraer la secuencia de palabras emitida por un locutor a partir de una señal acústica. Como paso previo a la implementación de un RAH, es necesario conocer los principios teóricos en los que se basa y decidir el enfoque de diseño que se va a adoptar.

Los sistemas de RAH pueden clasificarse según distintos criterios (adaptación al locutor, tamaño del vocabulario, objetivo del reconocimiento...) y existen varios enfoques a la hora de abordar el algoritmo de reconocimiento. Los métodos probabilísticos basados en el Teorema de Bayes son los que predominan hoy en día en el reconocimiento del habla [3].

El proceso de reconocimiento comienza con la captación de una señal de voz a través de un micrófono. Esta señal se procesa de forma que periódicamente se obtiene un vector de características $\mathbf{o}_t \in \mathbb{R}^D, t \in [1, T]$, siendo D el número de dimensiones del vector y T el número de observaciones. Mediante este proceso, conocido como extracción de características, se obtiene el conjunto de observaciones $\mathbf{O}$ que conforman los datos de entrada al sistema.

$$\mathbf{O} = \{\mathbf{o}_1, \ldots, \mathbf{o}_t, \ldots, \mathbf{o}_T\} \tag{1}$$

Tras el proceso de reconocimiento, la salida del sistema es una secuencia de N palabras.

$$\mathbf{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_k, \ldots, \mathbf{w}_K\} \tag{2}$$

El objetivo del sistema es conseguir la secuencia de palabras que mejor se ajusta a los datos de entrada, lo cual se expresa desde un punto de vista probabilístico como:

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmax}}\, P(\mathbf{W}|\mathbf{O}) \tag{3}$$

La probabilidad $P(\mathbf{W}|\mathbf{O})$ no puede calcularse directamente. Haciendo uso del Teorema de Bayes, (3) puede reformularse como:

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmax}}\, \frac{P(\mathbf{O}|\mathbf{W})\, P(\mathbf{W})}{P(\mathbf{O})} \tag{4}$$

$P(\mathbf{W})$ es la probabilidad de que la secuencia de palabras ocurra, la cual se obtiene a partir del modelo de lenguaje. $P(\mathbf{O}|\mathbf{W})$ es la probabilidad de que la secuencia de observaciones $\mathbf{O}$ se corresponda con la

secuencia de palabras pronunciadas $\mathbf{W}$, y viene dada por el modelo acústico. El término en el denominador, $P(\mathbf{O})$, no afecta a la maximización, por lo que puede eliminarse de (4), obteniéndose la fórmula fundamental del reconocimiento automático del habla:

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmax}} \{P(\mathbf{O}|\mathbf{W})\, P(\mathbf{W})\} \tag{5}$$

La Figura 1 representa los pasos del proceso de reconocimiento:

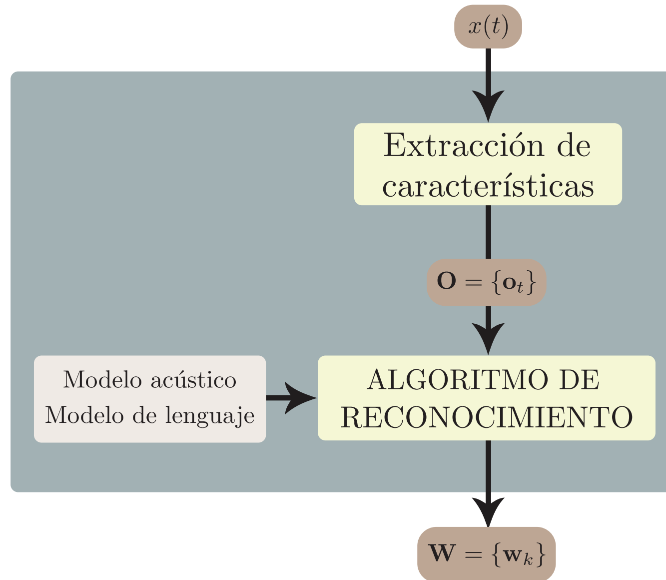


Figura 1: Esquema básico de un RAH

2.2 MODELOS OCULTOS DE MARKOV

La forma más habitual de modelar la producción del habla es mediante modelos ocultos de Markov o HMMs (Hidden Markov Models) [5]. Para comprenderlos fácilmente, es mejor comenzar viendo las cadenas de Markov.

Sean un conjunto de estados $S = \{s_1, s_2, \ldots, s_N\}$ con ciertas probabilidades de transición $A = \{a_{ij}\}$ entre ellos, los cuales producen unos determinados resultados observables $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, y una secuencia de variables aleatorias u observaciones $\mathbf{O} = \{\mathbf{o}_1, \ldots, \mathbf{o}_T\}$ que pueden tomar alguno de los valores en $\mathbf{X}$. La secuencia forma una cadena de Markov de orden 1 si cumple la Propiedad de Markov, esto es, si dado el estado actual, los estados pasados y los futuros son independientes.

$$P(S_{t+1} = s' \,|\, S_1 = s_1, \ldots, S_t = s) = P(S_{t+1} = s' \,|\, S_t = s) \tag{6}$$

Una cadena de Markov de orden m sería aquella en la que la probabilidad de ocurrencia de un estado depende de los m estados pasados, pero a partir de ahora se tomarán en consideración únicamente

modelos de orden 1. La Figura 2 representa una cadena de Markov con $N = 5$, probabilidad 1 de comenzar en el estado s_1 y distintas probabilidades de transición.

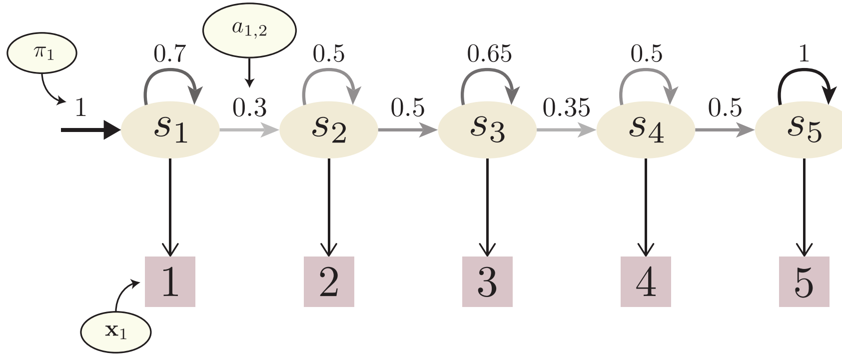


Figura 2: Ejemplo de cadena de Markov

Para tal cadena de Markov, la probabilidad de observar la secuencia $\mathbf{O} = \{1; 1; 2; 3; 3\}$ sería de:

$$P(\mathbf{O}) = \pi_1 \;\; a_{1,1} \;\; a_{1,2} \;\; a_{2,3} \;\; a_{3,3} = 6.825 \times 10^{-2} \tag{7}$$

A diferencia de las cadenas de Markov, donde observando los estados puede determinarse la verosimilitud de una secuencia, los estados de un HMM no son directamente observables sino que producen unos resultados observables u otros con una cierta probabilidad. De esta forma, la secuencia observada no se corresponde directamente con una secuencia de estados, sino que lo hace con una cierta probabilidad.

Un HMM se define como $\lambda = \{A, B, \Pi\}$, y tiene los siguientes parámetros:

- $S = \{s_1, \dots, s_N\}$: el conjunto de estados del modelo, siendo N el número de estados que lo forman.
- $B = \{b_1, \dots, b_N\}$: las probabilidades de distribución asociadas a cada uno de los estados. En el caso que aquí se trata la distribución de cada estado se modela como una mezcla de Gaussianas o GMM (*Gaussian Mixture Model*), cuya función de densidad de probabilidad (pdf, *Probability Density Function*) se define como la suma de un grupo de Gaussianas ponderadas por unos pesos. La verosimilitud de un vector de características $\mathbf{o_t}$ para un estado s_j es:

El Apéndice B detalla este cálculo y muestra su desarrollo como producto matricial.

$$b_j(\mathbf{o_t}) = \sum_{c=1}^{C} w_{j,c} \, \mathcal{N}(\mathbf{o_t}; \boldsymbol{\mu}_{j,c}, \boldsymbol{\Sigma}_{j,c}) \tag{8}$$

C es el número de Gaussianas en la mezcla, $w_{j,c}$ es el peso de la Gaussiana c, y $\boldsymbol{\mu}_{j,c}$ y $\boldsymbol{\Sigma}_{j,c}$, su media y covarianza, respectiva-

mente. En este proyecto, como ocurre habitualmente en RAH, se utilizan matrices de covarianza diagonales.

- $A = \{a_{i,j}\}$: probabilidades de transición entre cada pareja de estados, con $1 \leqslant i \leqslant N$ y $1 \leqslant j \leqslant N$, de forma que, en un instante t, $a_{i,j} = P(S_t = s_i | S_{t-1} = s_j)$.
- $\Pi = \{\pi_i\}$: las probabilidades iniciales, con $\pi_i = P(S_1 = s_i)$.

El ejemplo de la Figura 2 se convierte en un HMM si cada estado s_n lleva asociada una distribución Gaussiana de media n y varianza 0.5 (ver Figuras 3 y 4).

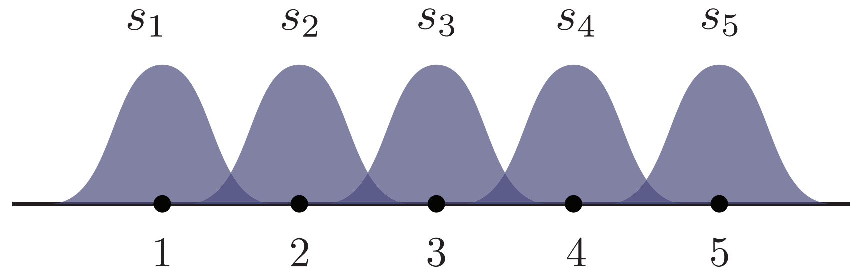


Figura 3: Probabilidades de distribución de los distintos estados

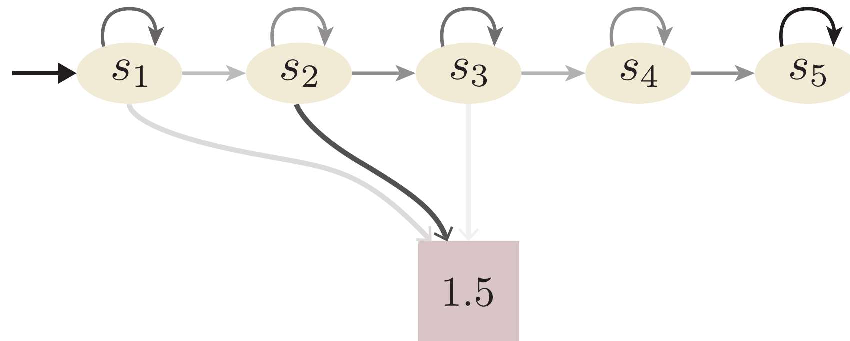


Figura 4: Probabilidades de observación de los distintos estados

En este caso la probabilidad de que una secuencia de entrada **O** haya sido producida por la secuencia de estados **S** se calcula de la siguiente manera:

$$P(\mathbf{O} \,|\, \mathbf{S})\, P(\mathbf{S}) = \pi_1\, b_1(\mathbf{o}_1) \prod_{i=2}^{5} a_{j(i-1),j(i)}\, b_{j(i)}(\mathbf{o}_i) \tag{9}$$

El algoritmo de reconocimiento busca la secuencia que maximiza esta probabilidad, lo cual es equivalente a resolver la ecuación (5). En el siguiente apartado se explica cómo llegar a esta solución.

En los sistemas de RAH el modelo de lenguaje es una cadena de Markov donde cada estado es una palabra, definiendo así las relaciones entre éstas y la probabilidad de las posibles secuencias. El modelo acústico es un HMM donde cada estado representa una unidad

de sonido, en este caso el fonema con contexto. Cada fonema se modela como tres estados, incluyendo así información sobre el fonema (o silencio) que lo precede y el que lo sigue dentro de una palabra. Los parámetros estadísticos de las redes utilizadas en RAH se calculan generalmente mediante una estimación de máxima verosimilitud (ML, *Maximum Likelihood*) usando el algoritmo iterativo EM (*Expectation Maximization*) [1] a partir de una base de datos con ejemplos de cada tipo de sonido.

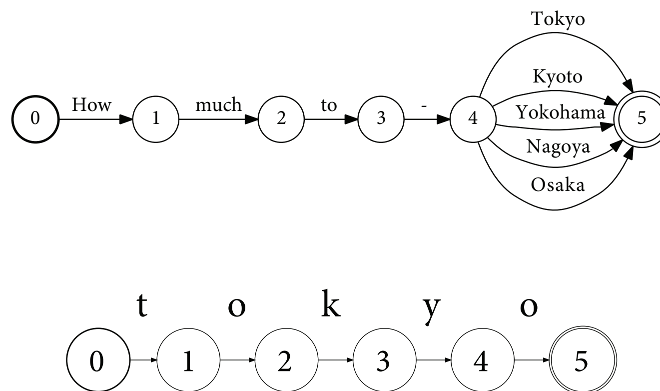


Figura 5: Ejemplo de red de palabras y de fonemas [2]

La Figura 5 muestra un ejemplo de una red de palabras y otro de una de fonemas. La red final que el algoritmo de reconocimiento recorre puede verse como una red por capas, producto de la composición de las redes de sendos modelos. De esta forma, dos estados de la capa inferior pueden compartir el mismo modelo estadístico, pero ser aún así distintos por pertenecer a palabras diferentes en la red superior. El algoritmo de reconocimiento recupera la secuencia de estados más probable, pero lo que interesa al usuario es la secuencia de palabras, por lo que una tabla asocia cada palabra a su último estado de la red inferior. Así, únicamente se muestran las palabras asociadas a estados dentro de la secuencia.

2.3 ALGORITMO DE VITERBI

A partir de los modelos acústico y de lenguaje se efectúa el proceso de reconocimiento, donde se calcula la secuencia de palabras que maximiza la ecuación (5).

Una solución de fuerza bruta podría ser, a partir de los datos de entrada, calcular la verosimilitud de todas las posibles secuencias de palabras, y obtener aquélla que la maximice. El problema de este enfoque, excepto para ejemplos muy sencillos (pocas palabras en el diccionario y secuencias de entrada cortas), es que la carga computacional de esta solución la hace inabordable.

Aprovechando la memoria finita de los HMMs usados en el problema, el algoritmo de Viterbi [5] permite reducir su complejidad resolviéndolo por partes. Éste recorre, a medida que van llegando nuevos datos de entrada, el diagrama de transiciones o diagrama de Trellis, calculando para cada estado la máxima verosimilitud y el estado desde el que se llega con ésta. Si un estado tiene dos o más transiciones de entrada, se puede mantener únicamente la secuencia que le llega con mayor probabilidad, ya que no hay forma de que las hipótesis descartadas superen en verosimilitud a la mantenida a partir de ese momento. A este proceso de eliminación de hipótesis se le llama purga.

La máxima verosimilitud se obtiene a partir de la siguiente ecuación:

$$P_t(j) = \max_i \{P_{t-1}(i)\, a_{ij}\, b_j(\mathbf{o_t})\} \qquad 2 \leqslant t \leqslant T; 1 \leqslant j \leqslant N \qquad (10)$$

Esta ecuación indica que la probabilidad del mejor camino que termina en el instante t y estado j se obtiene a partir del camino que con mayor probabilidad se propaga desde cada uno de los N estados en el instante $t-1$ hacia el estado j, multiplicándose ésta por la verosimilitud del vector de características $\mathbf{o}_t$ para el estado j en el instante t. Esta última verosimilitud es la misma para todos los caminos que confluyen en un mismo estado, por lo que la purga puede realizarse antes de este cálculo. En el instante $t = 1$ no puede aplicarse la ecuación (10), por lo que se multiplican las probabilidades iniciales π_j por las probabilidades de observación $b_j(\mathbf{o_1})$. En general se parte de un único estado inicial con probabilidad 1, lo cual simplifica el proceso de búsqueda.

La Figura 6 muestra el algoritmo de Viterbi para el ejemplo de HMM tratado en el apartado anterior. Las probabilidades acumuladas se muestran normalizadas a la máxima en ese momento.

Mientras hay datos de entrada los caminos se van propagando, repitiendo este cálculo iterativamente. Cuando llega la última observación, se calcula el estado final con mayor probabilidad acumulada y se recupera la secuencia de estados asociada a él buscando hacia atrás su estado de procedencia y el de los sucesivos estados recuperados. A esta búsqueda iterativa, ilustrada en la Figura 7, se la llama *backtracking*.

Para reducir los problemas de desbordamiento puede expresarse la ecuación (10) como la suma de los logaritmos de cada término:

$$\mathcal{L}_t(j) = \max_i \{\mathcal{L}_{t-1}(i) + \alpha_{ij} + \beta_j(\mathbf{o_t})\} \qquad 2 \leqslant t \leqslant T; 1 \leqslant j \leqslant N \quad (11)$$

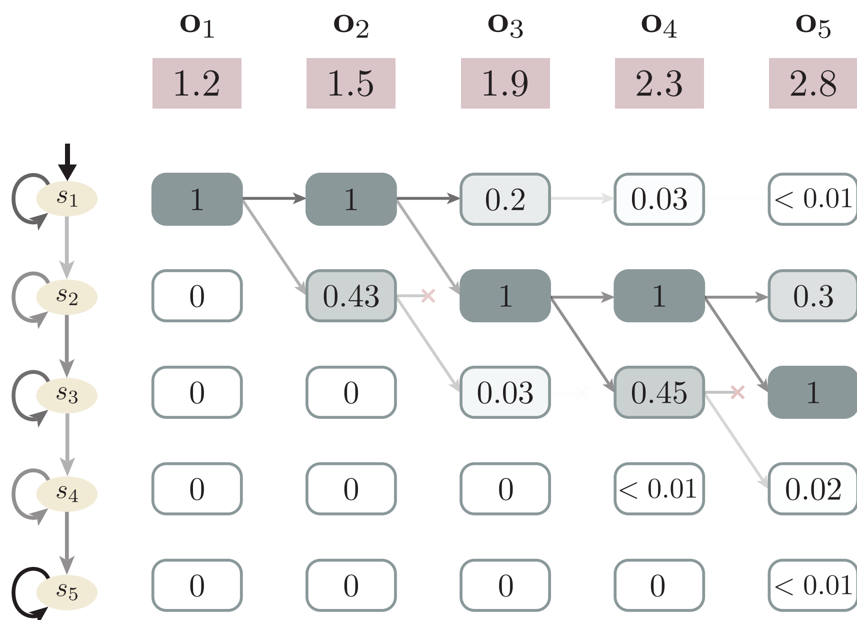


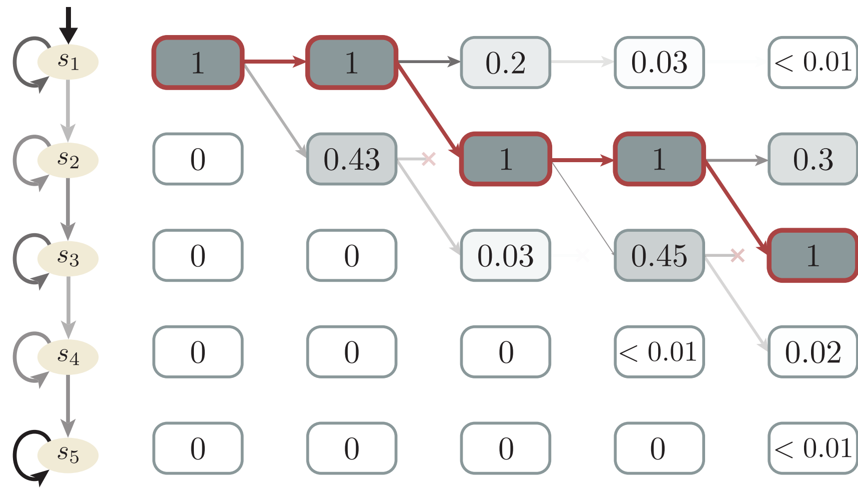
Figura 6: Ejemplo del algoritmo de Viterbi

2.4 búsqueda con tokens

Pese a que el algoritmo de Viterbi reduce la complejidad del problema de reconocer la secuencia de palabras más probable, en la mayoría de los casos el cálculo de la máxima probabilidad acumulada para todos los estados supone todavía una gran carga computacional. Las matrices de transiciones en RAH se caracterizan por tener un número de estados muy elevado con pocas transiciones de salida, por lo que calcular en cada frame temporal el camino hacia cada estado es costoso y en muchos casos innecesario.

Una variación del algoritmo mantiene en cada momento un determinado número de hipótesis activas o *tokens* (testigos) y realiza los cálculos únicamente para los caminos considerados posibles en ese momento. En cada paso de propagación, cada token se clona tantas veces como su número de transiciones de salida, actualizando su probabilidad acumulada con la probabilidad de transición y la probabilidad de observación del estado de destino. Si más de un token coincide en el mismo estado, se elige aquél con mayor probabilidad acumulada.

Si el número máximo de tokens M es igual a N los resultados de este algoritmo son iguales a los del de Viterbi. Sin embargo, suelen introducirse dos aproximaciones para reducir el espacio en memoria y la carga computacional requeridos por este algoritmo. Un parámetro llamado *beam* determina cuántas veces más pequeña ha de ser la probabilidad acumulada de un token frente al de la hipótesis más fuerte para considerarse como despreciable. Así, en cada iteración se elimi-



$$\hat{\mathbf{W}} = \{\mathbf{w}_1, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_2, \mathbf{w}_3\}$$

Figura 7: Búsqueda hacia atrás

nan los tokens despreciables (paso de *beam search*). Por otra parte, se suele mantener un número máximo de tokens $M < N$. Si tras una propagación hace falta un número mayor de tokens, se mantienen los M con mayor probabilidad acumulada.

El apéndice C profundiza en el algoritmo de búsqueda con tokens.

La Figura 8 ilustra el algoritmo con tokens, con un *beam* de 0,1.

$\mathbf{o}_1$	$\mathbf{o}_2$	$\mathbf{o}_3$	$\mathbf{o}_4$	$\mathbf{o}_5$
1.2	1.5	1.9	2.3	2.8

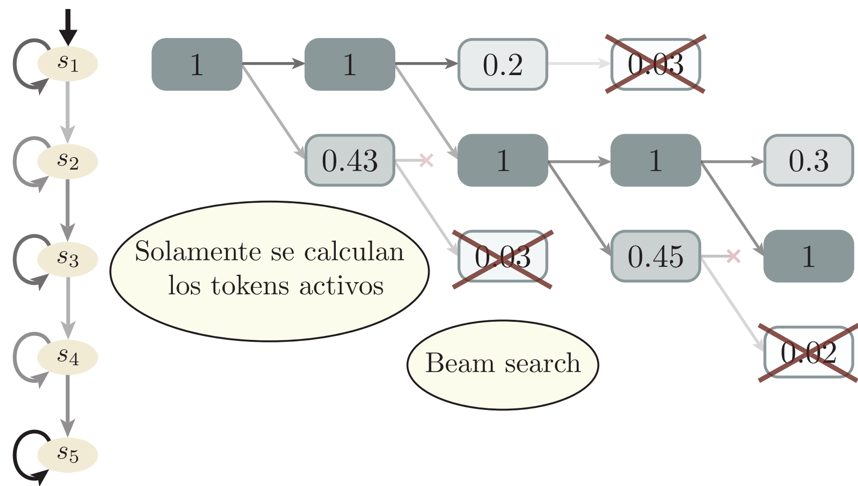


Figura 8: Ejemplo de búsqueda con tokens

Esta variación permite, además, presentar resultados parciales a mitad del algoritmo. Si en un determinado instante t todos los tokens provienen del mismo estado quiere decir que todas las hipótesis compartirán el mismo camino hasta $t-1$, por lo que los resultados hasta

ese instante pueden mostrarse ya. Los caminos asociados a los tokens se guardan en un buffer Φ, por lo que esta operación permite liberar espacio de él. La siguiente vez que se muestren resultados, se hará desde el momento $t+1$. En el caso de que el buffer se llene, se devuelve el camino asociado a la hipótesis más fuerte en ese momento, se limpia el buffer y se eliminan todos los tokens menos el de mayor peso.

3 IMPLEMENTACIÓN

3.1 INTRODUCCIÓN

Un buen sistema de RAH no necesita únicamente ser preciso, sino que en la mayoría de los casos la eficiencia de un reconocedor se mide como el número de errores en el reconocimiento en función del tiempo de ejecución, como puede verse en la Figura 9.

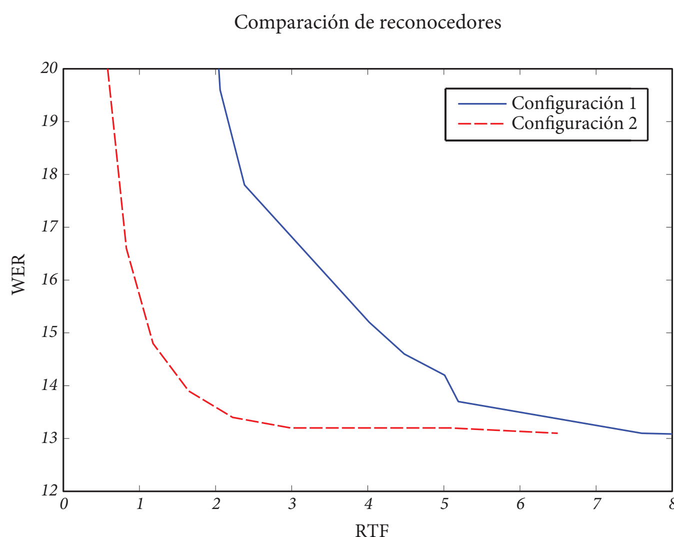


Figura 9: Número de errores frente a tiempo de ejecución[4]

De esta forma, el proyecto plantea la implementación de funciones en paralelo empleando GPUs y *CUDA C* como medio para mejorar los tiempos de ejecución. Asimismo, las estructuras de datos empleadas en este tipo de sistemas se caracterizan por ser de gran tamaño, pero con pocos elementos útiles dentro de ellas (por ejemplo, la matriz de transiciones A, representada en la Figura 10). Por lo tanto, el ahorro de espacio en memoria mediante estructuras de tipo *sparse* y la gestión de éstas es otra de las características principales del prototipo implementado.

El Apéndice A incluye detalles y ejemplos de programación en CUDA.

La estructura de un reconocedor suele basarse en bucles anidados, como por ejemplo en el paso de propagación (ver Algoritmo 1). Transformando el algoritmo a uno equivalente basado en operaciones matriciales y vectoriales puede verse qué partes de éste son susceptibles de ser paralelizadas.

En el Apéndice C puede encontrarse el algoritmo de reconocimiento basado en matrices.

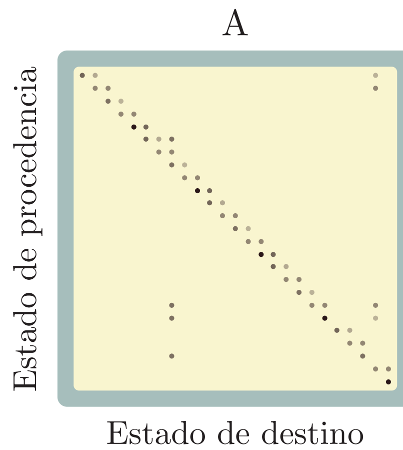


Figura 10: Ejemplo de matriz de transición

```
1 for i = 1 : n do ;                /* Para cada token activo */
2
3     for j = 1 : m do ;        /* Para cada estado de salida */
4
5         Calcular probabilidad acumulada Pa;
6         if (Pa > Pj) then Pj = Pa;
```

Algoritmo 1: Propagación por bucles anidados

El ejemplo anterior puede expresarse como una suma matricial (se trabaja en escala logarítmica) y una búsqueda de máximos por columnas.

La primera operación es claramente paralela, lo cual presenta gran oportunidad para mejorar los tiempos de ejecución, teniendo en cuenta que el número de tokens activos puede llegar a ser muy alto. Sin embargo, es más difícil implementar un algoritmo paralelo que realice la búsqueda de máximos, especialmente si se está trabajando con matrices sparse, en las cuales se incrementa la dependencia entre elementos.

Este compromiso se verá prácticamente en todos los pasos del proceso de paralelización, lo cual no permite determinar a priori el impacto de la nueva implementación en los tiempos de ejecución. En consecuencia, lo que el proyecto plantea es el estudio de esta solución mediante un prototipo con funciones en paralelo.

3.2 CONSIDERACIONES PREVIAS A LA IMPLEMENTACIÓN

Antes de comenzar a desarrollar el algoritmo es necesario decidir cómo trabajar con estructuras sparse. Existen librerías, como *CUSP*[1] o *cuSPARSE*[2] que implementan diversas funciones para realizar ope-

1 http://cusplibrary.github.io
2 http://developer.nvidia.com/cuSPARSE

raciones con matrices y vectores de este tipo. Pese a que éstas son probablemente más rápidas que una versión "hecha a mano" de las mismas, se trata de funciones generales optimizadas para el uso en resolución de sistemas de ecuaciones, por lo que el conocimiento del problema y la implementación de funciones a medida para éste puede resultar en una solución más eficiente que otra basada en el uso de librerías.

Una de las primeras tareas del proyecto ha sido la comparación de la librería cuSPARSE con estructuras y funciones propias para las operaciones básicas que el problema requiere. Tras realizar este estudio, se ha optado por la segunda estrategia, en base a las siguientes razones:

- No existe una librería que permita trabajar con estructuras sparse en escala logarítmica, lo cual acarrea problemas de rango, especialmente en el cálculo de las probabilidades de observación.
- Una de las principales operaciones del algoritmo de reconocimiento, el paso de propagación, consiste en multiplicar la probabilidad de cada token (elementos de un vector) por una fila de la matriz de transiciones. Esta operación no existe en las librerías, ni tampoco la conversión de vector a matriz necesaria para expresar este paso como un producto matricial. Además, la librería advierte que el producto entre dos matrices sparse es muy poco eficiente.
- El conocimiento del problema permite, al realizar ciertos pasos del algoritmo, adelantar trabajo de las siguientes operaciones.
- La solución basada en librerías, al estar pensada para otro tipo de aplicaciones, implica un trabajo no despreciable en gestión de tipos de datos y conversión de formatos para adecuarse al problema de RAH.

Tras definir las estructuras sparse a usar, se han agrupado distintas variables en estructuras y distintos pasos en funciones, de forma que el programa principal queda legible y estructurado (la Figura 11 ilustra su funcionamiento básico). En él se distinguen las siguientes partes, algunas de las cuales se detallarán en los apartados siguientes:

Las estructuras utilizadas en el proyecto están especificadas en el Apéndice D.

- *Declaración de variables y asignación de valores*: Al comienzo del programa es necesario declarar las distintas variables que se van a usar y reservar espacio en memoria para ellas, en función de los parámetros del problema. Algunos de estos parámetros, como el tamaño del buffer o el beam, están definidos como macros, mientras que otros dependen del problema de reconocimiento concreto que se vaya a tratar. Los archivos que contienen

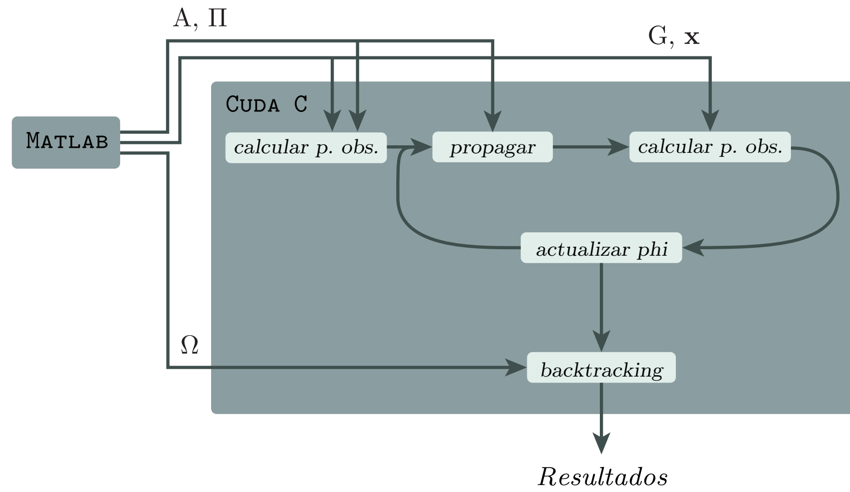


Figura 11: Funcionamiento del reconocedor

los datos de entrada al problema (la matriz de transiciones, las mezclas de Gaussianas, las observaciones y el diccionario), todos adecuados al problema mediante un programa escrito en Matlab, contienen el resto de parámetros necesarios, como el número de estados o el número máximo de transiciones desde un estado. Las distintas estructuras se inicializan mediante distintas funciones en el host o en la GPU según sea necesario, y se copian los operandos necesarios a la GPU.

- *Tratamiento de la primera observación*: Tras leer la primera observación, y con el vector de tokens inicializado con las probabilidades iniciales Π, se ponderan éstas con las probabilidades de observación de cada estado.

- *Bucle de reconocimiento*: Mientras hay nuevos datos de entrada con observaciones se ejecuta este bucle, que comienza con la propagación de los tokens (paso que incluye, además, la purga, el beam search y la normalización de éstos), cuyas probabilidades acumuladas se actualizan después con las probabilidades de observación. Finalmente, se actualiza la matriz Φ a la vez que se comprueba si está llena o si todos los tokens provienen del mismo estado, en cuyo caso se llama a la función de backtracking, la cual recupera la secuencia asociada al token más probable desde el último frame recuperado hasta el actual y muestra por pantalla las palabras que ésta representa.

- *Backtracking final*: Cuando no hay más datos de entrada, se llama por última vez a la función de backtracking, la cual devuelve la secuencia asociada al token final más probable almacenada en el buffer desde la última llamada a la función.

3.3 PROPAGACIÓN

Esta parte del algoritmo, situada al comienzo del bucle de iteración, se encarga de calcular los tokens activos en el siguiente frame temporal.

Se utilizan las estructuras `tok`, de tipo `VSparse`, para guardar los tokens activos, `B`, de tipo `FSparse`, para guardar los resultados de la generación de nuevos tokens, y `A`, de tipo `Trans`, la cual almacena las transiciones de salida de cada estado.

```
1 B = repmat (tok, 1, N) + A;        /* Cálculo de transiciones */
2 [tok, i_prev] = max (B, [], 1)';                    /* Purga */
3 max_tok = max (tok);
4 tok = tok - max_tok;                        /* Normalización */
5 tok(tok < beam) = −∞;                        /* Beam Search */
6 active = find (tok > −∞);     /* Índices de tokens activos */
```

Algoritmo 2: Propagación de tokens

3.3.1 *Cálculo de tokens en el siguiente frame*

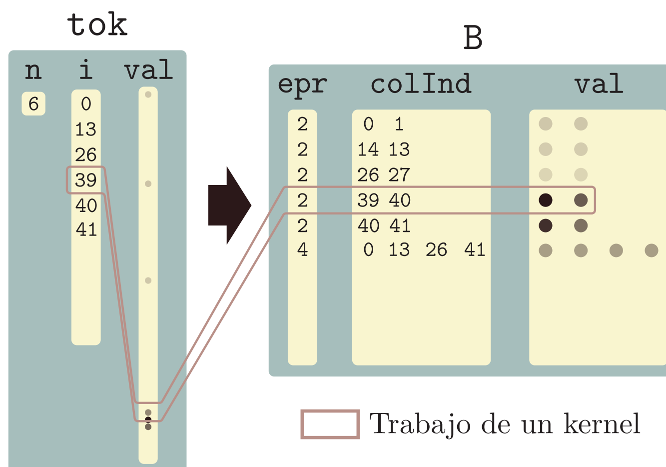


Figura 12: Cálculo de los nuevos tokens: resultado

La primera operación que realiza esta función es la generación de tokens en el siguiente frame temporal, (primera línea del Algoritmo 2) donde cada hipótesis crea un nuevo token por cada una de las transiciones de salida del estado donde se encuentra. Como se ha explicado, este paso puede verse como la suma de la probabilidad acumulada de cada token activo a todos los elemento de su fila correspondiente en la matriz `A`, aunque en este caso en la matriz resultante, `B`, las filas

con elementos distintos de cero están apiladas al principio (ver Figura 12). El índice de la fila a la que corresponden está guardado en el vector de tokens.

El kernel (Figura 13) se ejecuta de forma que hay un hilo por cada uno de los `n` tokens activos, el cual se encarga de leer todas las probabilidades de salida de su estado y generar los nuevos tokens. La matriz `A`, como se ha dicho, guarda los valores no nulos y sus columnas en sendos vectores, y el índice del primer elemento de cada fila en otro vector. Por lo tanto, restando los índices de dos filas consecutivas puede saberse cuántas transiciones de salida tiene un estado.

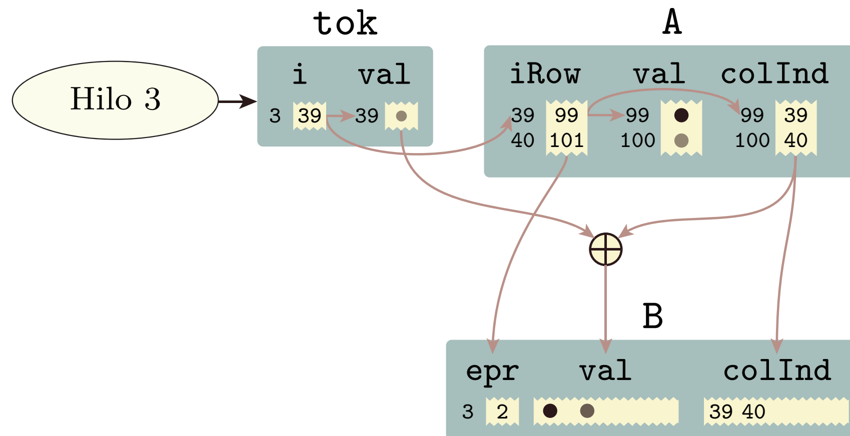


Figura 13: Cálculo de los nuevos tokens: kernel

3.3.2 *Purga*

El siguiente paso es volver a rellenar el vector de tokens eligiendo el más probable de cada estado de destino (línea 2 del Algoritmo 2). Primero es necesario poner todos los elementos de `tok`$\rightarrow$`val` a $-\infty$, de lo cual se encarga un kernel con tantos hilos como elementos en éste.

La búsqueda del token más probable en cada estado tiene una dependencia intrínseca entre elementos. Sin embargo, puede aprovecharse el hecho de que todos los tokens en cada fila de `B` se han propagado desde el mismo estado, por lo que éstos no necesitan compararse entre sí. En este kernel hay un hilo por cada columna en `B` que recorre los `n` estados de procedencia. Cuando encuentra un token activo, si su probabilidad es mayor que la que hay en ese estado en `tok`$\rightarrow$`val` (de ahí la puesta a $-\infty$), guarda su valor y estado de procedencia. La Figura 14 muestra el trabajo conjunto de los hilos para purgar los tokens de cierto estado de procedencia.

A medida que un hilo recorre los estados de procedencia va guardando la mayor probabilidad que ha encontrado. Como el siguiente paso va a ser buscar la máxima probabilidad encontrada para posteriormente realizar el beam search, de esta manera se reduce el espacio

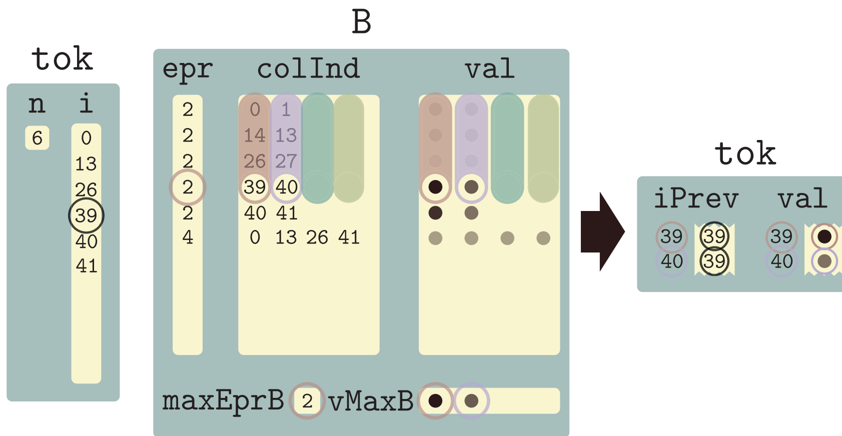


Figura 14: Algoritmo de purga: cuarto estado de procedencia

de búsqueda. El primer hilo guarda también el máximo número de transiciones que ha encontrado (`maxEprB`) para saber entre cuántos elementos habrá que realizar la búsqueda. La Figura 15 muestra los resultados de la operación.

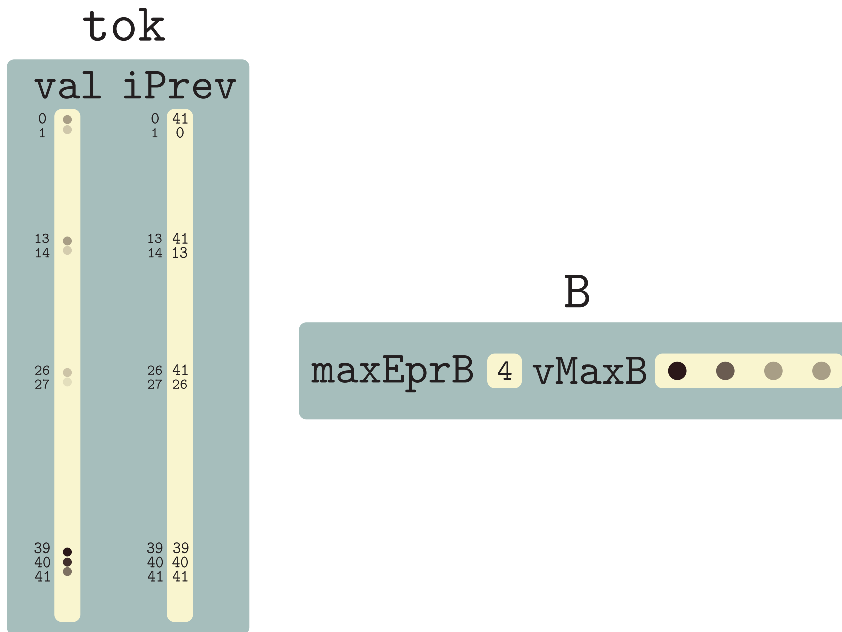


Figura 15: Algoritmo de purga: resultado

3.3.3 *Búsqueda del máximo por reducción*

La reducción es una técnica común en algoritmos paralelos que sirve para optimizar operaciones que pueden expresarse como un árbol de decisiones, tales como algoritmos de búsqueda o la suma de los elementos en un vector (ver Figura 16). Se trata de un algoritmo iterativo donde un hilo realiza la operación entre dos elementos, de forma

que en cada iteración se reduce a la mitad el número de elementos. En la última iteración queda un elemento, producto de la operación entre todos los del vector.

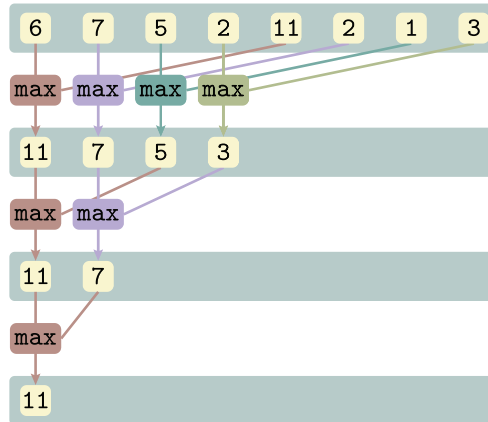


Figura 16: Búsqueda del máximo por reducción

Éste es el procedimiento que se sigue para hallar la probabilidad máxima entre los tokens activos. Cada bloque de n hilos puede buscar el máximo entre $2n$ elementos. Debido a que trabajar con la memoria compartida de bloque es mucho menos costoso que acceder directamente al vector, al principio del kernel cada hilo lee dos elementos de `vMaxB` (o, si uno de ellos está fuera de rango, escribe el elemento neutro en la memoria compartida) para después hacer la reducción entre estos elementos. De esta forma, si el tamaño del vector es mayor que el doble del tamaño de bloque se obtienen resultados parciales, uno por bloque, que son guardados en un vector auxiliar, `vAuxMaxB`, repitiendo el proceso de reducción (alternando estos dos vectores) hasta que quede un único elemento (ver Figura 17).

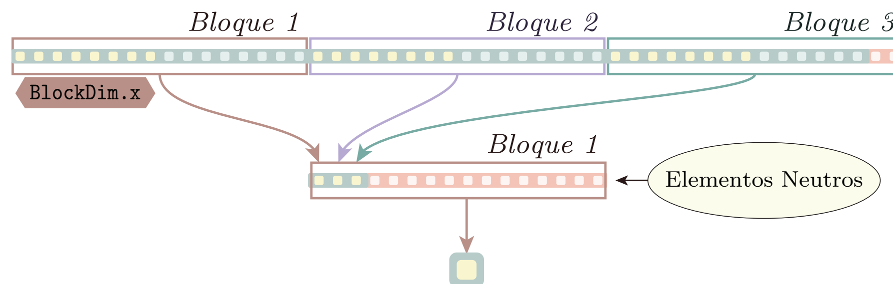


Figura 17: Reducción de varios bloques

Cuando se detecta que el número de bloques necesario es 1 la dirección de destino que se le pasa a la función será la del valor máximo, `maxVal`. El valor guardado en esta dirección es usado por otro kernel para normalizar todos los elementos en `tok`→`val` Este kernel, al igual que la puesta a $-\infty$ es ejecutado por tantos hilos como elementos hay en este vector.

3.3.4 *Actualización de índices y número de tokens activos*

Antes de dar por concluida la propagación de tokens al siguiente frame temporal es necesario realizar el beam search y actualizar los índices del vector `tok`. En el caso de que haya más valores distintos de $-\infty$ que el número máximo de tokens se toman los de mayor peso. No se ha encontrado una función paralela que realice esta operación, por lo que se ha programado como una función secuencial que se ejecuta en la CPU.

Esta función recorre el vector `tok→val`, el cual se ha copiado previamente al host, y cuando encuentra un token activo que no esté por debajo del beam guarda su estado en `tok→i` e incrementa el valor en `tok→n`. A la vez, va guardando la posición y el valor mínimo encontrado hasta el momento. Si se alcanza el número máximo de tokens, será ese valor el que haya que superar para entrar en el vector de tokens. Cuando un token cumple esta condición, se vuelve a hacer una búsqueda del mínimo entre los tokens guardados hasta el momento.

3.4 Cálculo de las probabilidades de observación

Tras leer cada una de las observaciones se llama a la función `get_pObs`, la cual calcula la probabilidad de observación de los estados activos. Cada una de éstas se puede calcular de la siguiente forma:

El desarrollo completo del cálculo se halla en el Apéndice B

$$\mathbf{v}_j = G_j \cdot \begin{bmatrix} \mathbf{o}_t^2 \\ \mathbf{o}_t \\ 1 \end{bmatrix}, \quad b_j(\mathbf{o}_t) = \sum_{c=1}^{C} e^{v_{j,c}}, \tag{12}$$

La estructura `G` contiene las matrices y vectores necesarios para el cálculo. La observación de cada frame se guarda en el vector `x` con el formato $[\mathbf{o}_t^2; \mathbf{o}_t; 1]$, mientras que la estructura `Gauss` contiene todas las posibles matrices G_j, una por cada GMM, apiladas de forma que se puedan obtener todos los vectores $\mathbf{v}_j$ con un único producto matriz-vector. Estos vectores de resultados parciales también se guardan en un vector, `pExp`. Las distintas posibilidades de observación se encontrarán al final del cálculo en el vector `pObs`, apiladas al inicio, tras lo cual servirán para actualizar las probabilidades acumuladas de los tokens activos.

```
1 b = eval_st (X_t, active, G);                /* Cálculo de p. obs. */
2 tok(active) = tok(active) + b;              /* Actualización de tok */
```

Algoritmo 3: Actualización con las probabilidades de observación

3.4.1 *Inicialización de* `gMask` *y* `fetch`

Como no todos los estados están activos en un determinado momento, multiplicar todas las filas de la matriz de parámetros de las Gaussianas por el vector de observaciones sería una pérdida de tiempo considerable. La estructura `G` tiene, por tanto, una máscara `gMask` que indica qué filas deben multiplicarse. Un kernel con un hilo por token se encarga de activar los elementos de ésta correspondientes a los estados activos en ese frame (ver Figura 18).

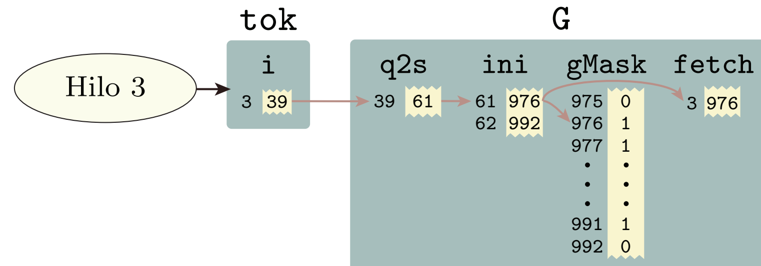


Figura 18: Inicialización de `gMask` y `fetch`: kernel

Cada estado está modelado por una GMM, cuyo número de Gaussianas no tiene por qué ser fijo. Además, varios estados pueden compartir una misma GMM. La tabla `q2s` establece una correspondencia entre cada estado y su mezcla, mientras que el vector `ini` contiene la fila inicial de cada GMM en la matriz de parámetros, por lo que la diferencia entre dos elementos consecutivos de este vector da el número de Gaussianas en la mezcla.

Cada hilo guarda también en el vector `fetch` la posición de la primera Gaussiana que va a utilizar, ya que la probabilidad de observación estará guardada en esa posición en el vector `pExp`. De esta forma se calcula la probabilidad de observación de cada GMM una única vez, aunque ésta corresponda a varios estados activos.

3.4.2 *Multiplicación con máscara*

El paso siguiente es realizar la multiplicación matriz-vector. El kernel encargado de esta operación tiene un hilo por cada fila de la matriz `G`. Cada hilo de un bloque carga un elemento del vector en la memoria compartida y, si la máscara está activada en su posición, realiza la multiplicación fila-vector de un número de elementos igual al tamaño de bloque. Si el vector es más grande, los hilos volverán a repetir esta operación hasta completar los productos fila-vector. La Figura 19 ilustra este proceso.

En esta función puede haber bloques que carguen el vector `x` sin que ninguno de los hilos tenga que hacer después el producto matriz-vector. Debido al número variable de Gaussianas por GMM y al formato de ésta, no puede conocerse a priori el número de filas a multi-

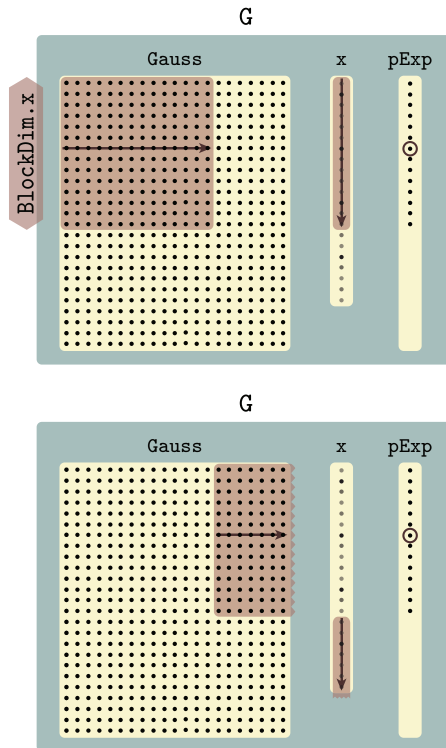


Figura 19: Multiplicación matriz-vector con máscara: trabajo de un bloque

plicar en esta operación. Esto hace necesario pasar por todas las filas de `G`.

3.4.3 *Suma y normalización*

La operación con la que se obtienen las probabilidades de observación es la suma de la exponencial de los resultados de cada Gaussiana dentro de una GMM. Para evitar problemas de rango, se normalizan todos los resultados antes de hacer la exponencial, obteniendo el máximo de entre todos los resultados usando el mismo algoritmo de reducción empleado en el paso de propagación, con un hilo por cada una de las `tot` Gaussianas. Posteriormente otro kernel se ocupa de sumar los resultados de cada GMM, con un hilo por cada una de ellas. A medida que va haciéndolo, prepara la máscara para la siguiente iteración del bucle de reconocimiento, desactivando los elementos de `gMask` correspondientes a esa GMM. Finalmente, hace el logaritmo de la suma para dejarlo todo en esta escala.

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i >= n) return; // Un hilo por GMM

int jIni = ini[i];  // Gaussiana inicial de la GMM
if(!gMask[jIni]) return;  // Sólo GMMs activas
int jFin = ini[i+1];  // Gaussiana inicial de la siguiente
```

```
float res = 0.;
for(int j = jIni; j < jFin; j++) {
  res += exp(pExp[j] - *max); // Suma
  gMask[ii] = false;  // Desactivación de flags
}
pExp[jIni] = log(res);
```

Listing 1: Kernel con suma de resultados

3.4.4 *Actualización de las probabilidades acumuladas*

En el último kernel hay un hilo por cada token activo que recupera los resultados en `pExp` y los escribe en su posición correspondiente en `pObs`, de forma que al final los `tok→n` primeros elementos de éste contienen las probabilidades de observación.

Al finalizar la función `get_pObs` el programa principal llama a otro kernel que se ocupa de actualizar las probabilidades acumuladas de los tokens activos, sumándoles las probabilidades de observación (ver Figura 20).

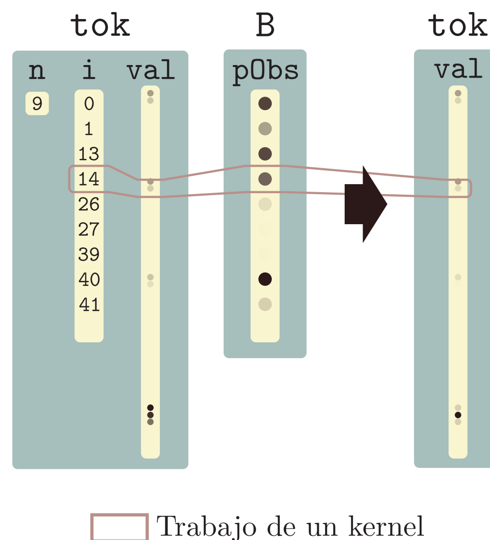


Figura 20: Actualización de los tokens con las probabilidades de observación

3.5 RECUPERACIÓN DE RESULTADOS

3.5.1 *Actualización del buffer* Φ

Tras actualizar los tokens con las probabilidades de transición, se llama a la función `update_phi`, la cual registra los estados de procedencia de los tokens activos en ese frame en una matriz de índices, llamada `P` en el código. En esta matriz cada fila corresponde a un fra-

me temporal, la columna de un elemento representa el estado en el que está el token y el valor es su estado de procedencia. Se trata de un buffer circular (la primera fila es la siguiente a la última) donde dos punteros, `tIni` y `tFin`, indican la primera y la última fila efectivas.

1 [tok, i_prev] = `max` (B, [], 1)′; `/* Purga */`

2 ...

3 active = `find` (tok $> -\infty$);

4 ...

5 Φ(active, t) = i_prev(active); `/* Actualización de Φ */`

Algoritmo 4: Actualización de Φ

Un kernel con un hilo por token activo se encarga de rellenar la fila número `P`→`tFin`, tras lo cual se incrementa este puntero circularmente.

```
unsigned int j = blockIdx.x*blockDim.x + threadIdx.x;
if (j >= nTok) return; // Un hilo por token activo

int pos = colsPerRow * tFin + j;

int col = iTok[j]; // Estado del token -> columna de P
colIndP[pos] = col;

int valpos = iPrevTok[col]; // Estado previo -> valor en P
valP[pos] = valpos;

if (i == 0) eprP[row] = n;  // Número de elementos en la fila
else {
  int col0 = tok->i[0]; // Comparación con el primer elemento
  if (valpos != tok->iPrev[col0]) *(P->eq) = false;
}
```

Listing 2: Kernel con actualización de `P`

A continuación, otro kernel comprueba si todos los tokens provienen del mismo estado. Antes de su llamada, se inicializa una variable booleana, `eq`, a `false`. Cada hilo del kernel compara el estado previo de un token activo con el del primero. Si son distintos, desactiva el flag, de forma que éste al final indica si todos los estados son iguales.

La función `update_phi` devuelve este flag al programa principal, el cual activa una llamada a la función `backtracking`. También se comprueba si al incrementar `tFin` éste ha alcanzado a `tIni`, en cuyo caso el buffer está lleno y hay que mostrar igualmente los resultados parciales para liberar espacio. En ambos casos, tras esta llamada se actualizan los punteros de `P`.

3.5.2 *Algoritmo de bactracking*

La función `backtracking` es llamada en los casos citados anteriormente y cuando no quedan más observaciones por leer. El algoritmo, que va recuperando la secuencia de estados más probable desde el más reciente hasta el más antiguo, varía ligeramente entre estos casos, por lo que una variable indica a la función en cuál de ellos se encuentra.

La función comienza calculando el número de estados a recuperar, tras lo cual hay que guardar en `seq[nPhi]` el estado final más probable.

```
int nPhi = P->tFin - P->tIni;
if (nPhi <= 0) nPhi += seq->maxT;
if (why == 0) nPhi --;
```

Listing 3: Número de estados a recuperar

Si todos los tokens provienen del mismo estado se decrementa `nPhi` ya que hay que recuperar la secuencia únicamente hasta el frame anterior. En éste se ha propagado un solo estado, cuyo índice está en el vector `tok→iPrev` en cualquiera de las posiciones guardadas en `tok→i`. Basta, por tanto, consultar el estado previo del primer token activo (por simplicidad) para conocer el estado final más probable en ese frame.

En los otros dos casos se recupera la secuencia hasta el frame actual y hay que buscar el estado más probable en ese momento. La función `max_value_ind` realiza esta búsqueda por reducción devolviendo el estado final más probable, el cual se guarda en `seq`. Aunque el algoritmo de reducción es el mismo (se devuelve un resultado por bloque y se alterna entre dos vectores hasta tener un único resultado final), el kernel al que llama esta función trabaja con vectores de índices, los cuales sirven para consultar y comparar los valores de un un vector de tipo `VSparse`.

A partir de este token se va extrayendo del buffer `P` su secuencia de estados asociada. Un bucle iterativo se encarga de rellenar los elementos de `seq` desde `nPhi − 1` hasta `0` mediante la llamada a la función `prev_state`. El valor de `P` en la posición correspondiente al último estado recuperado (la fila se corresponde con el frame y la columna con el estado), será su estado de procedencia (ver Figura 21).

De esta forma, `prev_state` ejecuta un kernel en el cual cada hilo se ocupa de una columna de `P` en esa fila o frame temporal. Si el último estado recuperado se corresponde con esa columna significa que forma parte del camino más probable, por lo que escribe el valor en esa posición en la secuencia.

```
cudaMemcpy(seq->h_seq, seq->d_seq, n * sizeof(int),
  cudaMemcpyDeviceToHost);  // Copia de seq a la CPU
```

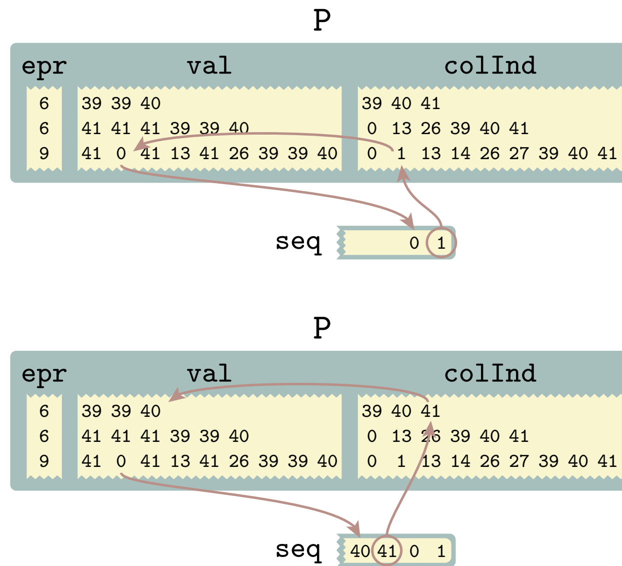


Figura 21: Búsqueda del estado anterior en la secuencia

```
int st, stPrev;
stPrev = seq->stFin;  // Último estado recuperado

for (int j = 0; j < n; j++) {
  st = seq->h_seq[j];
  if(st == stPrev) continue; // Comprobar si cambia de estado
  if( strlen(dict[st]) > 0 )  // Si lleva palabra asociada
    printf("%s ", dict[st]);
  stPrev = st;
}
fflush(0);

seq->stFin = st;  // Guardar último estado recuperado
```

Listing 4: Número de estados a recuperar

Finalmente, se imprime la secuencia de palabras. En el caso de buffer lleno no se tiene en cuenta el último estado, porque la siguiente vez que se haga backtracking será el primero. La función `print_seq` es una función secuencial que trabaja en CPU (los datos necesarios como la secuencia se copian antes desde la GPU). Ésta recorre la secuencia y, para cada estado, comprueba la tabla-diccionario `dict` y muestra por pantalla su palabra asociada, en caso de que la haya y si no es igual al estado anterior (un estado puede durar varios frames). Una variable de la estructura `seq` guarda el último estado aparecido entre distintas llamadas a la función `print_seq`, para evitar el error de sacar dos veces la misma palabra cuando no corresponde.

4 RESULTADOS

A medida que se ha desarrollado el prototipo de reconocedor, éste se ha ido probando con distintos modelos estadísticos y datos de entrada para comprobar su correcto funcionamiento.

Al no ser posible acceder a las tarjetas gráficas durante la ejecución, el proceso de depurado de las funciones en paralelo se ha realizado copiando los resultados al host e imprimiéndolos por pantalla.

- Para depurar las distintas funciones a nivel bajo se ha usado un modelo creado para tal propósito, con datos de entrada artificiales de una dimensión y cinco estados posibles que siguen una distribución Gaussiana. Un modelo tan simple dista mucho de una aplicación real, pero sirve para seguir el proceso de reconocimiento paso a paso y conocer los valores de las variables en todo momento, comprobando el funcionamiento básico del sistema.
- Un modelo simple pero real se ha usado para depurar los cálculos de las probabilidades de observación, lo cual requiere vectores de entrada multidimensionales, y el algoritmo de backtracking con diccionario, sacando palabras en vez de estados. Las secuencias de palabras de este modelo son series de dígitos en inglés, por lo que la gramática tiene un tamaño reducido y es posible todavía mostrar resultados intermedios y variables por pantalla, aunque es más difícil seguir su valor en todo momento. Este ejemplo también ha servido para comprobar el funcionamiento del programa que adapta el formato de datos de otros reconocedores, como el HTK, al usado por el prototipo.
- Finalmente, una gramática que modela preguntas de geografía se ha usado para comprobar el funcionamiento del prototipo con estructuras de grandes dimensiones (lo cual es útil para verificar la coordinación entre varios bloques de hilos para las distintas funciones en paralelo) y para comparar los resultados del prototipo con otros reconocedores.

4.1 ESTUDIO DE TIEMPOS

Para el estudio de tiempos se ha reconocido una frase con el tercer modelo de los citados en el apartado anterior. La grabación dura 3,6 segundos, por lo que un tiempo de reconocimiento menor se considera como"reconocimiento en tiempo real". Las simulaciones se han ejecutado en el un nodo, "`voz08`", del clúster del Grupo de Tecnologías de las Comunicaciones de la Universidad de Zaragoza. Se ha utilizado una CPU *Intel Xeon E5645 @2.40 GHz* y una GPU *nVidia GeForce GTX 660 Ti* .

4.1.1 *Comparación del rendimiento con otros reconocedores*

Debido a que el prototipo ejecuta unas operaciones muy distintas de otros reconocedores secuenciales, es difícil comparar los tiempos de ejecución de manera justa. Asimismo, la falta de adaptación de los datos del prototipo al formato usado por otros reconocedores hace difícil una evaluación sistemática de la tasa de error frente al tiempo. Por tanto, se ha decidido comparar el tiempo de ejecución del prototipo con el de otros reconocedores en función del número medio de tokens activos por frame, lo cual indica cómo de eficiente es la gestión de tokens. En el tiempo de ejecución no se ha incluido la inicialización de las distintas variables y la carga de los modelos estadísticos.

Los reconocedores utilizados para la comparación han sido el HTK, el cual es un software de RAH ampliamente utilizado, y el reconocedor del Laboratorio de Tecnologías del Habla de la Universidad de Zaragoza, en dos configuraciones distintas que aquí llamaremos KTree y WFST debido a los algoritmos que emplean.

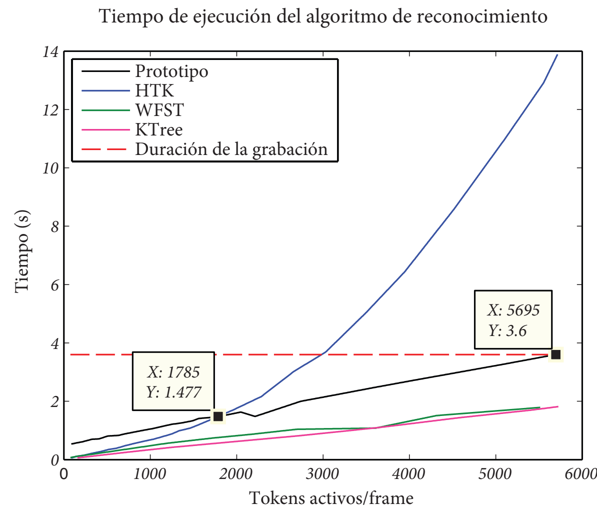


Figura 22: Comparación de gestión de tokens

Las Figuras 22 y 23 muestran esta comparación, en la que puede observarse que, aunque para un número reducido de tokens HTK es más rápido, el prototipo consigue gestionar menos tokens reconociendo la frase sin errores, y más dentro del límite del reconocimiento en tiempo real. El otro reconocedor es más rápido en sus dos configuraciones, aunque también tiene fallos en el reconocimiento para un número de tokens reducido. Estos reconocedores tienen más parámetros que permiten optimizar su funcionamiento y usan distintas técnicas para reducir los tiempos de ejecución, alejándolos del algoritmo de Viterbi canónico. Por otra parte, el formato del modelo estadístico del prototipo es más compacto, lo cual agiliza su carga. La inicialización de variables del prototipo dura en torno a 0.6 segundos, mientras que

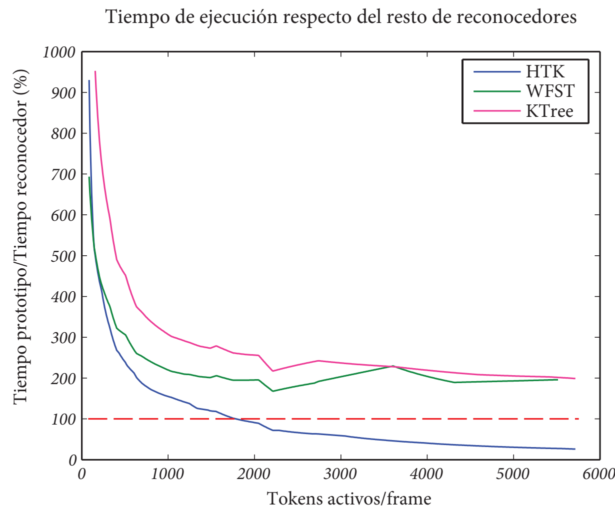


Figura 23: Comparación de gestión de tokens

al resto de reconocedores les cuesta entre 5 segundos (HTK) hasta superar la decena (KTree).

4.1.2 *Distribución de tiempos*

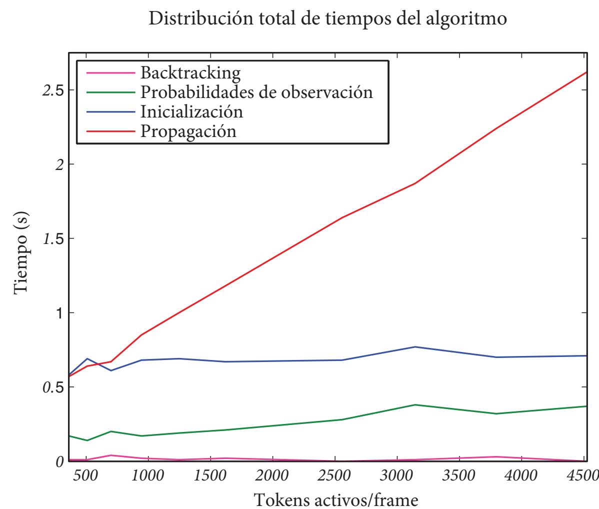


Figura 24: Distribución de tiempos

Las Figuras 24 y 25 muestran la distribución de tiempos dentro del programa principal. El proceso más costoso de todas las operaciones es el de propagación, cuyo tiempo se ha desglosado en las Figuras 26. y 27 La actualización de los índices del vector `tok` tras la propagación se había diseñado inicialmente como un kernel de un solo hilo, el cual es más lento que la misma función en la CPU pero ahorra la transferencia de datos entre host y device. Tras comprobar el impacto

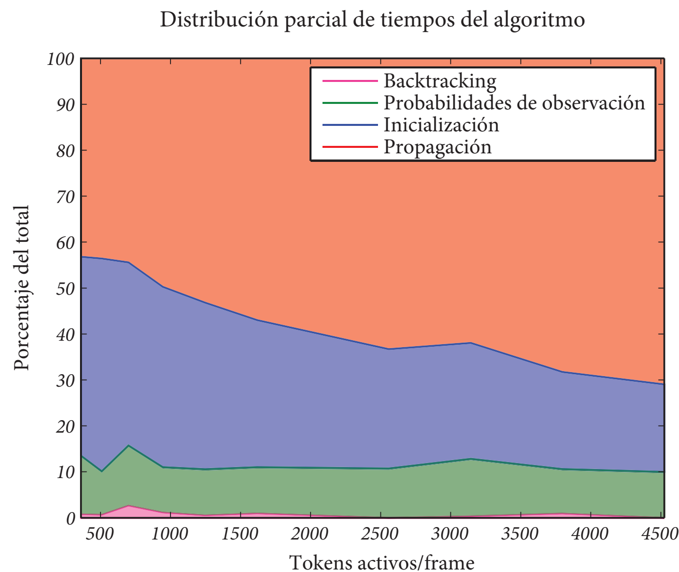


Figura 25: Distribución de tiempos

de esta función en el tiempo de ejecución del programa, se ha implementado la misma función en la CPU, reduciendo considerablemente el tiempo de ejecución, por lo que se ha mantenido en el código final.

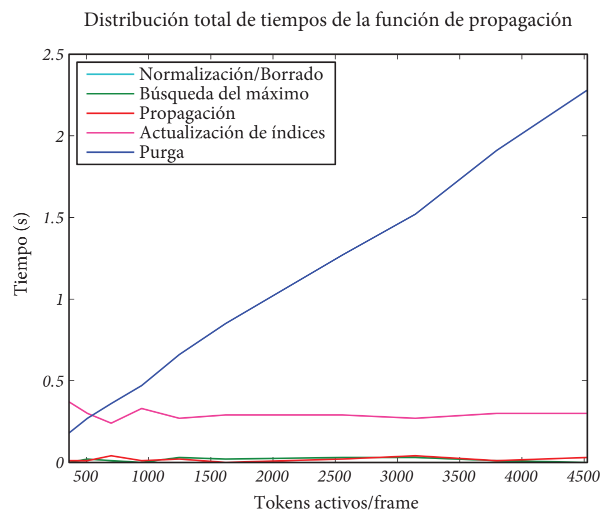


Figura 26: Distribución de tiempos de la función de propagación

La función que merece más la pena optimizar tras este cambio es la purga de tokens, la cual es una búsqueda del mejor token que ha llegado a cada estado y cuyo tiempo crece con el número de tokens activos. Aunque es hasta cierto punto paralelizable, el kernel tiene un bucle que termina con la sincronización de todos los hilos en cada iteración, lo cual ralentiza su funcionamiento. Podría estudiarse si el realizar la purga en la CPU (lo cual podría aprovecharse para realizar a la vez la búsqueda de la máxima probabilidad acumulada) aceleraría el proceso.

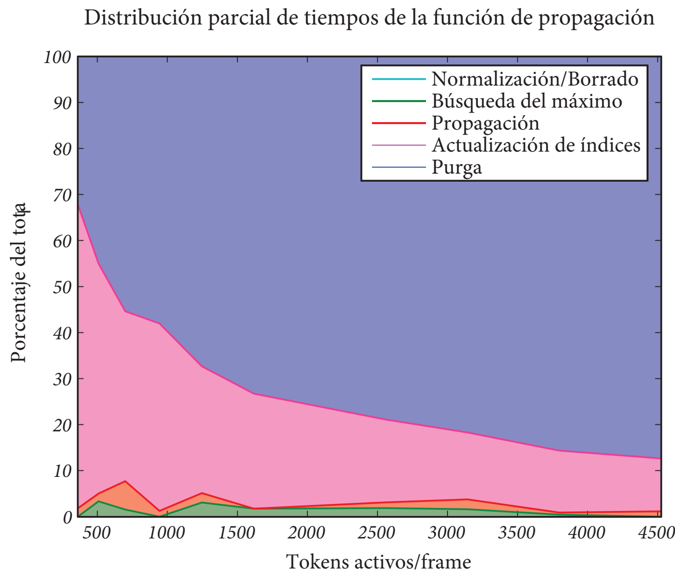


Figura 27: Distribución de tiempos de la función de propagación

4.1.3 *Rendimiento del cálculo de probabilidades de observación*

El tiempo de ejecución del cálculo de las probabilidades de observación en el prototipo sí que puede compararse fácilmente con el de otros programas, ya que puede definirse el número de observaciones a calcular. El algoritmo del prototipo realiza una multiplicación con máscara por lo que el porcentaje de GMMs activas influye en los tiempos de ejecución de forma proporcional, como puede verse en la Figura 28. Ésta muestra el tiempo requerido para calcular las probabilidades de observación de distinto número de GMMs respecto del total en función del número de frames temporales para los que se calcula.

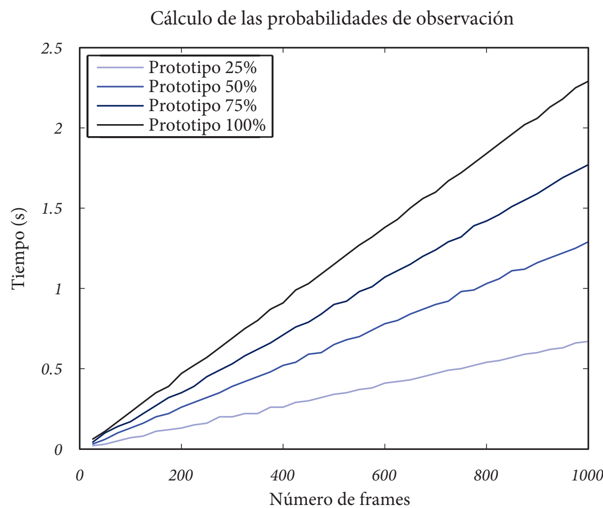


Figura 28: Impacto del número de frames

La figura 29 muestra la comparación entre los tiempos de ejecución del algoritmo del prototipo y los del reconocedor del Laboratorio de Tecnologías del Habla. Puede verse que el prototipo es más rápido en este cálculo cuando tiene que calcular las probabilidades de observación de muchos estados distintos.

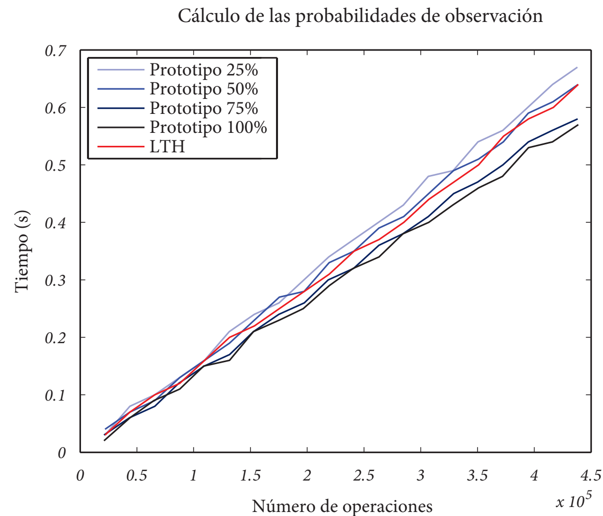


Figura 29: Comparación del cálculo de las probabilidades de observación

El tiempo de ejecución de este algoritmo está penalizado por el hecho de no conocer los índices de las Gaussianas que se van a calcular. En su lugar, esta información se halla en una máscara. Por tanto, las operaciones se realizan para todas las Gaussianas, y la máscara indica en cuáles de ellas pueden evitarse ciertos cálculos. Este enfoque de diseño se eligió, como en varios momentos del proceso de desarrollo, entre varias alternativas, sin que hubiera una solución claramente óptima.

Pese a que el impacto de este cálculo en el tiempo total del proceso de reconocimiento no es muy grande, sería interesante optimizarlo, ya que es una operación que puede servir para otras aplicaciones. Una alternativa que se planteó y que podría probarse consiste en, una vez conocidos los índices de las GMM que se van a calcular, construir una tabla con éstos, hallar el número total de Gaussianas que representan y construir una tabla con los índices de éstas, de forma que para el cálculo matriz-vector haría falta solamente un hilo por cada fila que se fuera a calcular, y éste sabría de dónde tomar sus operandos y dónde escribir su resultado.

5 CONCLUSIONES

5.1 RESUMEN DEL PROYECTO Y ANÁLISIS DE OBJETIVOS

El trabajo desempeñado durante el proyecto aparece desglosado en un cronograma en la Figura 30. Las tareas desempeñadas pueden clasificarse en tres tipos distintos:

- Recopilación de información sobre las herramientas utilizadas durante el proyecto, tales como CUDA y las librerías cuSPARSE, las distintas implementaciones existentes de estructuras sparse o el algoritmo de Viterbi con tokens.
- Implementación de algoritmos y funciones ya existentes y adaptación al prototipo, como por ejemplo la implementación matricial del bucle de reconocimiento en Matlab y C o el cálculo matricial de las probabilidades de observación.
- Diseño de funciones y estructuras características de la solución planteada, tales como las funciones paralelas, el método de gestión de datos o las distintas estructuras creadas.
- Medida de tiempos del prototipo y de otros reconocedores, análisis de resultados y cambios finales en el código a partir de éstos.

Se ha conseguido desarrollar un prototipo de reconocedor automático del habla con funciones en paralelo que funcione correctamente, lo cual cumple el objetivo principal del proyecto, ya que permite analizar su rendimiento frente a otros de funcionamiento secuencial. Se ha diseñado e implementado el funcionamiento del reconocedor con el consiguiente análisis de su funcionamiento y sus posibilidades de paralelización, mientras que los modelos acústicos y de lenguaje ya estaban generados y entrenados.

También se ha desarrollado una filosofía de gestión de tokens creada específicamente para el problema, y se ha comparado con el uso de las librerías sparse paralelas disponibles para CUDA, eligiéndose el primer enfoque. Esta experiencia con este tipo de librerías no es únicamente válida para el prototipo desarrollado sino que el análisis de sus puntos fuertes y débiles puede ser útil para futuros proyectos. Cabe remarcar que una vez realizado el esfuerzo inicial de cambiar la filosofía del algoritmo y de tomar las distintas decisiones de diseño, el trabajo de introducir mejoras en el código para seguir aumentando su rendimiento es considerablemente menor.

Además de funcionar, bajo ciertas condiciones el prototipo mejora los tiempos de ejecución de un reconocedor ampliamente usado como el HTK. Estas mejoras, además, pueden ser mayores con GPUs de mayor rendimiento, o explotando otros niveles de paralelismo, lo cual no sucede de igual forma con el software secuencial.

5.2 DESARROLLO EN EL FUTURO

Aunque se pueden introducir posibles optimizaciones en las distintas partes del código, lo más productivo es intentar mejorar los tiempos de ejecución de aquéllas que más tiempo toman, como la propagación y en concreto la función de purga. También es interesante seguir mejorando el cálculo de las probabilidades de observación, ya que éste es útil para aplicaciones más allá del RAH. Para ambas funciones, se han propuesto alternativas en el Capítulo 4. También pueden estudiarse variaciones en el proceso de reconocimiento, como la propagación hacia atrás, o incluso enfoques completamente distintos como el reconocimiento basado en redes neuronales, las cuales se pueden implementar directamente en paralelo.

Con un cierto trabajo de adaptación puede aplicarse esta filosofía a otros tipos de paralelismo tales como el uso de CPUs con varios procesadores o de clústeres, utilizando el código o los algoritmos ya creados para conseguir mejores tiempos de ejecución en máquinas más potentes.

Finalmente, el objetivo principal a partir de este momento es el uso del conocimiento ganado durante el proyecto y, si procede, de las funciones desarrolladas, en futuros proyectos relacionados con el RAH para conseguir un software de reconocimiento más rápido y robusto.

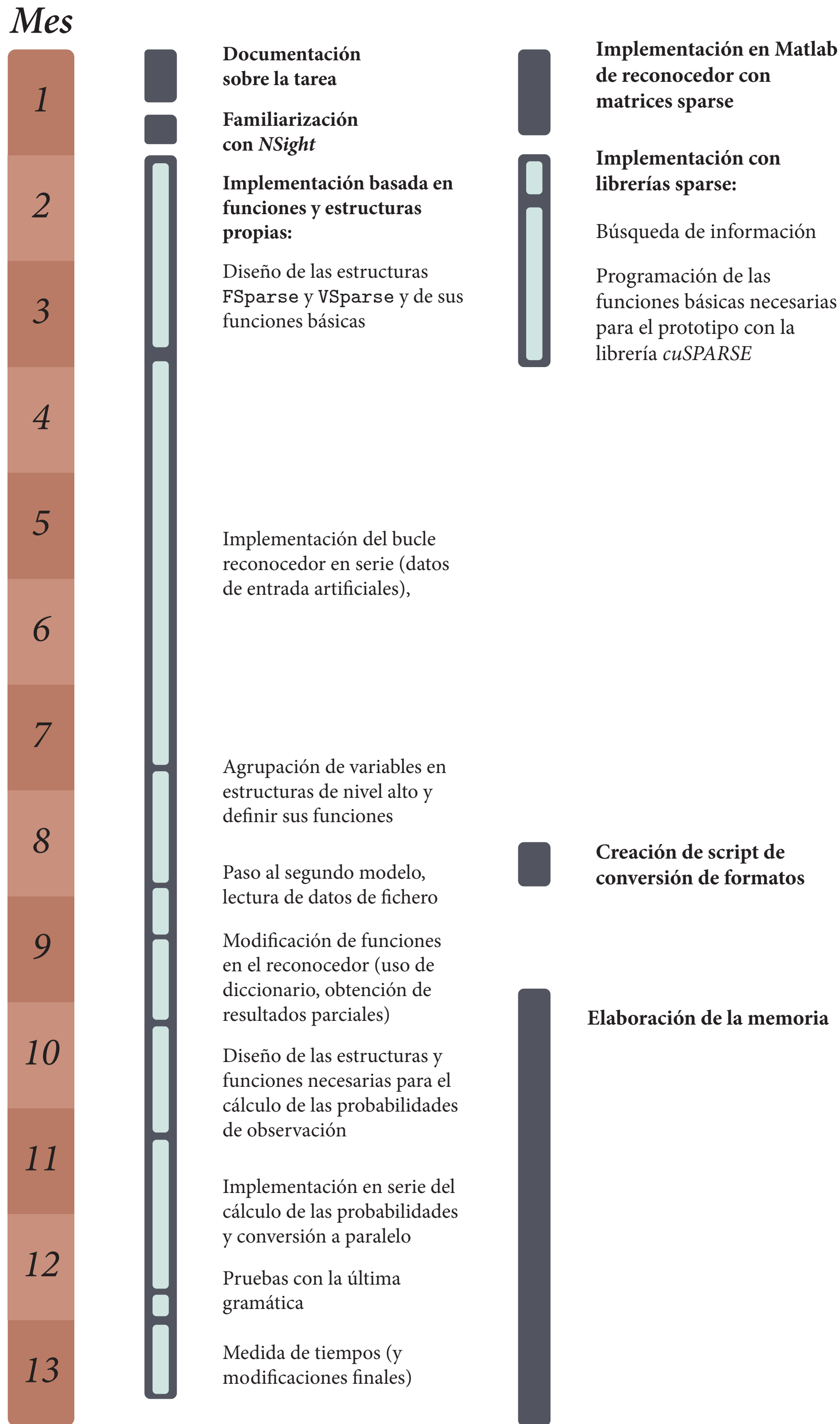


Figura 30: Cronograma del proyecto

A

CONCEPTOS BÁSICOS DE CUDA

En este apéndice se profundiza en los conceptos básicos de la programación en paralelo y de CUDA[1], en los cuales está basada la implementación del Reconocedor Automático del Habla tratado en este proyecto.

A.1 HISTORIA DE LA PROGRAMACIÓN EN PARALELO

La ley de Moore es la observación empírica del aumento de la densidad de transistores en un microprocesador, la cual se ha duplicado, desde la formulación de la ley en 1965, cada 18 a 24 meses (ver Figura 31). Este aumento en la densidad de transistores permitió, durante muchos años, aumentar la frecuencia de reloj de los procesadores, lo cual se tradujo directamente en mejoras en el rendimiento del software (el mismo programa es más rápido en un procesador con una frecuencia de reloj mayor). Sin embargo, la frecuencia de trabajo es directamente proporcional a la energía consumida por un chip (y consecuentemente a la energía disipada). De esta forma, pese a que la densidad de transistores sigue creciendo, las mejoras en la frecuencia de reloj han dejado de seguir esta tendencia (Figura 32).

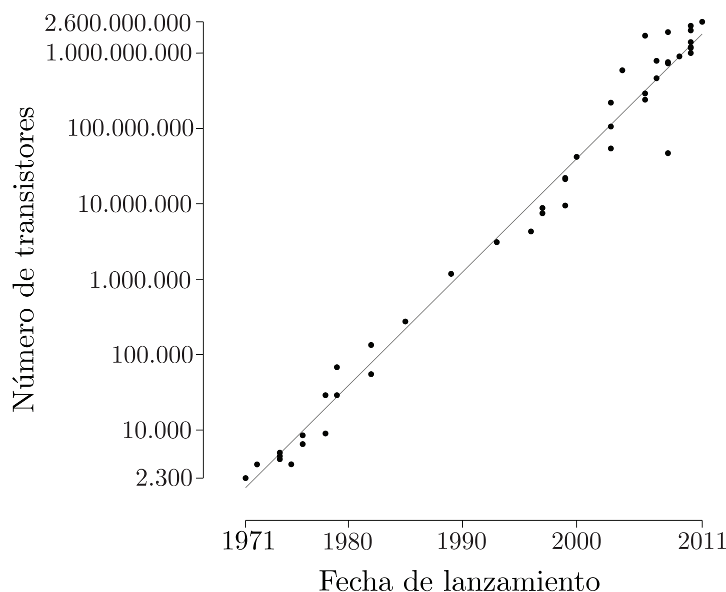


Figura 31: Número de transistores en distintos microprocesadores

Tradicionalmente, el software se ha escrito para ser ejecutado en serie, es decir, una instrucción se ejecutaba tras finalizar la anterior.

1 http://www.nvidia.com/object/cuda_home_new.html

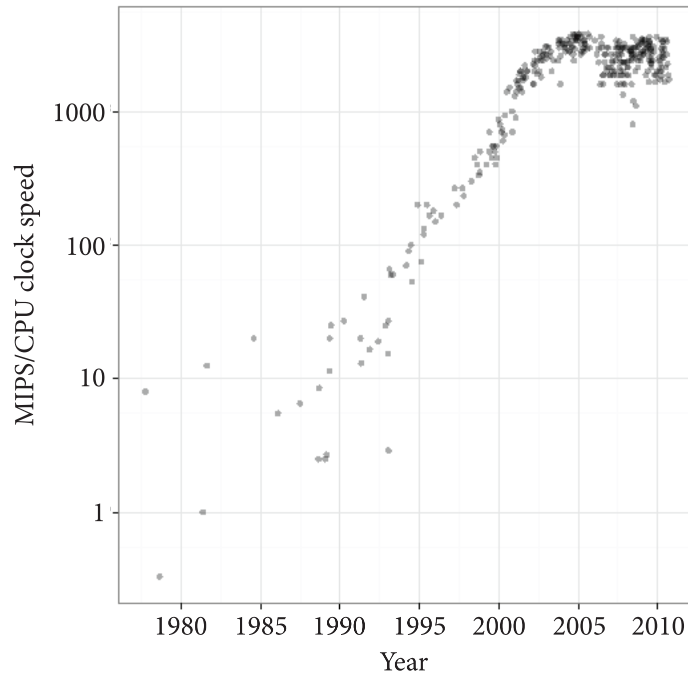


Figura 32: Evolución de la frecuencia de reloj

En contraposición, en la programación en paralelo se busca dividir las tareas a ejecutar en problemas independientes para poder resolverlos en varios procesos que se ejecutan a la vez. Aunque el paralelismo no es un concepto nuevo en la informática, los esfuerzos en avanzar en este modelo de programación se han aumentado en los últimos años con el objetivo de seguir consiguiendo mejoras en el rendimiento.

Hay varios niveles donde puede explotarse el paralelismo, desde los bits (una ALU capaz de hacer sumas de 16 bits acabará antes determinadas tareas que una que solamente procese 8 bits) o las instrucciones (segmentación de instrucciones en los microprocesadores) hasta llegar a otras soluciones como los procesadores multinúcleo, los clústers o los grids. El enfoque empleado por este proyecto ha sido el empleo de GPUs (*Graphic Processing Units*) consistentes en una serie de procesadores, lentos en comparación con una CPU, capaces de ejecutar el mismo código a la vez (ver Figura 33). Estas tarjetas surgieron con el propósito de optimizar el procesamiento digital de imágenes, en el cual la mayoría de las tareas tratan cada píxel de forma independiente.

Debido a que las GPUs comenzaron a usarse para aplicaciones distintas de aquellas para las que se habían concebido, se crearon entornos de desarrollo como CUDA, los cuales permiten acceder a la memoria y al conjunto de instrucciones de las GPUs mediante extensiones de lenguajes de programación estándar como C, C++ o Fortran.

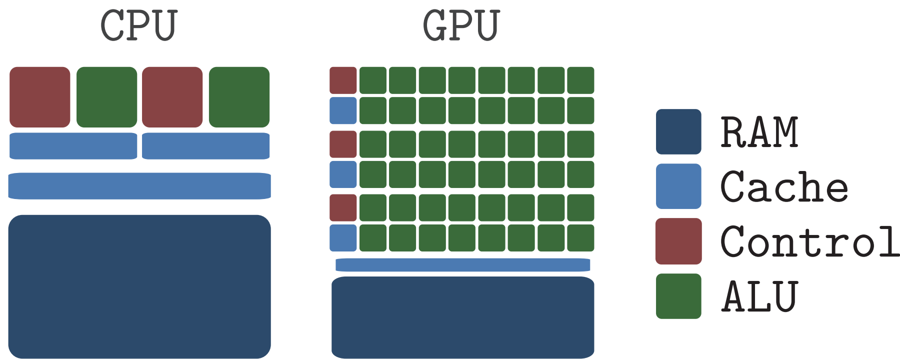


Figura 33: Arquitectura de una CPU y de una GPU

En este proyecto se utiliza la extensión *CUDA C/C++* para el desarrollo de las funciones en paralelo del reconocedor.

A.2 INTRODUCCIÓN A CUDA C

Las aplicaciones desarrolladas en CUDA C están basadas en un modelo de programación *host+device* heterogéneo donde en un único programa las partes en serie se ejecutan en el *host* o CPU mientras que las partes en paralelo lo hacen en el *device* o GPU. A la GPU se accede mediante funciones o *kernels* cuya llamada crea un conjunto de hilos paralelos que ejecutan el código de la función. El conjunto de los hilos que se crean para ejecutar el kernel, llamado grid, se divide a su vez en bloques de hilos, todos del mismo tamaño (ver Figura 34). Tanto un grid como sus bloques pueden distribuirse en hasta 3 dimensiones, lo cual simplifica el direccionamiento de memoria en ciertas aplicaciones como el procesamiento digital de imágenes.

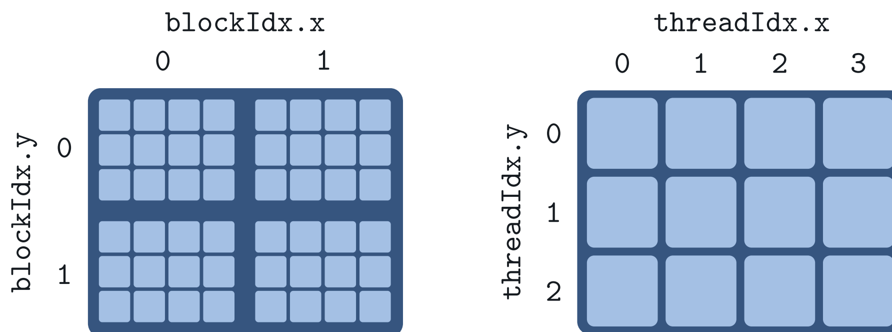


Figura 34: Ejemplo de grid y bloque en un kernel

Los tamaños de grid y bloque se definen antes de llamar a la función. Así, una llamada a un kernel quedaría de la siguiente manera:

```
// Código en host
...
// Llamada al kernel
dim3 blockDim(bx, by, 1);
dim3 gridDim(gx, gy, 1);
```

```
kernel<<< gridDim, blockDim>>>(args);
// Código en el host
...
```

Listing 5: Número de estados a recuperar

Cada hilo tiene unos índices mediante los cuales se puede conocer a qué bloque pertenece y qué posición dentro del mismo ocupa, lo cual sirve para calcular posiciones de memoria o gestionar diversas operaciones de control.

```
int ix = blockIdx.x * blockDim.x + threadIdx.x;
int iy = blockIdx.y * blockDim.y + threadIdx.y;
```

Listing 6: Número de estados a recuperar

Dentro de un bloque, los hilos pueden cooperar mediante el uso de memoria compartida o instrucciones de sincronización (ningún hilo dentro del bloque avanza hasta que todos hayan ejecutado dicha instrucción).

A.3 EJEMPLO: SUMA DE VECTORES

Las operaciones matriciales tales como sumas o productos, donde el cada elemento del resultado es independiente del resto, suelen producir grandes mejoras en los tiempos de ejecución al implementarse como funciones en la GPU. El siguiente código muestra un kernel que toma sendos elementos de dos vectores, los suma y guarda el resultado en la misma posición de un tercer vector.

```
__global__ void add( int *a, int *b, int *c ) {
  int tid = blockIdx.x; // sumar los elementos en esta posición
3 if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Listing 7: Número de estados a recuperar

Este kernel es ejecutado cuando es llamado por una aplicación, la cual le pasa los argumentos `a`, `b` y `c` y define las dimensiones de grid y de bloque. Los kernels pueden recibir argumentos por valor y por referencia, pero los argumentos por referencia tienen que ser direcciones de la memoria device. De esta forma, al principio del siguiente código se reserva memoria en la GPU para los vectores y se les da valor copiándolos desde la CPU.

```
#define N 10
int main( void ) {
  int a[N], b[N], c[N]; // vectores en CPU
  int *dev_a, *dev_b, *dev_c; // vectores en GPU
5
```

```
    // reservar memoria en GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
10
    // rellenar los vectores "a" y "b" en la CPU
    for (int i=0; i<N; i++) {
      a[i] = -i;
      b[i] = i * i;
15  }

    // copiar los operandos a la GPU
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice
         );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice
         );
20
    // sumar los operandos en GPU
    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copiar el resultado de GPU a CPU
25  HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
        cudaMemcpyDeviceToHost ) );
    // mostrar los resultados
    for (int i=0; i<N; i++) {
      printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
30
    // liberar la memoria reservada en GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
35  return 0;
}
```

Listing 8: Número de estados a recuperar

B

CÁLCULO DE LAS PROBABILIDADES DE OBSERVACIÓN

B.1 INTRODUCCIÓN

El proceso de reconocimiento se va realizando a medida que llegan nuevos datos de entrada al sistema. Éstos son vectores multidimensionales, resultado del proceso de características de un frame temporal de una señal de audio. A la secuencia de observaciones en el proceso de reconocimiento se la denomina con el nombre de $\mathbf{O}$:

$$\mathbf{O} = \{\mathbf{o}_1, \ldots, \mathbf{o}_t, \ldots, \mathbf{o}_T\} \tag{13}$$

El modelo acústico tiene un conjunto de posibles estados o unidades sonoras básicas, S, cada uno con una distribución estadística.

$$S = \{s_1, \ldots, s_j, s_N\} \tag{14}$$

Estos parámetros estadísticos permiten determinar la verosimilitud $b_j(\mathbf{o}_t) = P(\mathbf{o}_t|s_j)$, es decir, que una observación se corresponda con un estado. En general en RAH se utilizan modelos de mezcla de Gaussianas o GMMs, cuyas funciones de densidad de probabilidad son sumas ponderadas de las de varias distribuciones normales. Así, la probabilidad de observación del vector $\mathbf{o}_t$ para el estado s_j quedaría:

$$b_j(\mathbf{o_t}) = \sum_{c=1}^{C} w_{j,c}\, \mathcal{N}(\mathbf{o_t}\,; \boldsymbol{\mu}_{\mathbf{j,c}}, \boldsymbol{\Sigma}_{\mathbf{j,c}}), \tag{15}$$

donde C es el número de Gaussianas en la mezcla, $w_{j,c}$ es el peso de la Gaussiana c y $\boldsymbol{\mu}_{j,c}$, $\boldsymbol{\Sigma}_{j,c}$ son la media y la covarianza de la Gaussiana, respectivamente, y $\mathcal{N}$ es la probabilidad de observación para una distribución normal. Para un vector de observación de D dimensiones, ésta se calcula habitualmente de la siguiente manera:

$$\mathcal{N}(\mathbf{x}\,; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}\,|\boldsymbol{\Sigma}|^{1/2}}\, e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})'\,\boldsymbol{\Sigma}^{-1}\,(\mathbf{x}-\boldsymbol{\mu})} \tag{16}$$

B.2 EXPRESIÓN DEL CÁLCULO COMO PRODUCTO MATRICIAL

En este proyecto, como ocurre habitualmente en RAH, se utilizan matrices de covarianza diagonales, lo cual permite expresar el exponente de la ecuación (16) como:

$$\frac{1}{2}\sum_{i=1}^{D}\frac{(x_i - \mu_i)^2}{\sigma_i} = \sum_{i=1}^{D} a_i x_i^2 + b_i x_i + c_i, \tag{17}$$

lo cual es un producto escalar entre dos vectores, uno dependiente de la distribución y otro del vector de observaciones.

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} & c \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}^2 \\ \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{g} \cdot \chi \tag{18}$$

Subiendo al exponente el peso de la Gaussiana en la mezcla y la parte lineal de (16) es posible calcular de esta manera, en escala logarítmica, cada uno de los elementos a sumar en (15). Apilando los coeficientes de las distintas Gaussianas en una matriz, pueden obtenerse todos estos elementos en una multiplicación matriz-vector

$$G\chi = \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_C \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}^2 \\ \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{v} \tag{19}$$

La suma de las exponenciales de cada uno de los elementos ν_c del vector $\mathbf{v}$ es equivalente al resultado de (15).

$$\sum_{c=1}^{C} e^{\nu_c} = b(\mathbf{x}) \tag{20}$$

También es posible apilar las Gaussianas de más de una GMM para, posteriormente, elevar y sumar distintas partes del vector resultante para obtener las probabilidades de observación de distintos estados.

Esta transformación, además de representar una reducción en el coste computacional del cálculo de las probabilidades de observación, lo expresa como una operación fundamentalmente matricial, lo cual abre la puerta a una posible ganancia todavía mayor mediante el uso de algoritmos paralelos.

C ALGORITMO DE RECONOCIMIENTO

El presente apéndice detalla el algoritmo de reconocimiento empleado en este proyecto, sin entrar en los detalles de su implementación ni en los del cálculo de las probabilidades de observación, ya detallados en el Apéndice B.

C.1 CONSIDERACIONES PREVIAS

Generalmente, el algoritmo de reconocimiento de un RAH está basado en bucles iterativos. Como paso previo al planteamiento de la paralelización del algoritmo, éste se escribió en forma matricial en código Matlab, el cual es más sencillo de implementar que C y tiene múltiples opciones para representar los resultados gráficamente.

Todos los cálculos del algoritmo están en escala logarítmica, ya que se trabaja con probabilidades muy pequeñas que podrían salirse de rango en escala lineal. Aunque las matrices y los vectores con los que se trabaja en la implementación son de tipo *sparse* (donde únicamente se guardan los índices y los valores de los elementos no nulos), aquí no se entrará en tales detalles de implementación.

Los datos de entrada al algoritmo son los siguientes:

- $\mathbf{X} = \{\mathbf{X}_0, \ldots, \mathbf{X}_t, \ldots, \mathbf{X}_T\}$: vectores D-dimensionales con las observaciones en cada *frame* temporal.
- $A = \{\alpha_{i,j}\}$: probabilidades de transición, en escala logarítmica, entre cada pareja de estados, con $1 \leqslant i \leqslant N$ y $1 \leqslant j \leqslant N$, de forma que, en un instante t, $\alpha_{i,j} = \log(P(s_t = s_i | s_{t-1} = s_j))$.
- $\Pi = \{\pi_i\}$: vector de N elementos con la probabilidad inicial de cada estado, con $\pi_i = \log(P(s_1 = s_i))$
- G: conjunto de parámetros que define el modelo estadístico de cada estado, como la correspondencia entre un estado y una GMM o los parámetros de ésta. La función encargada de calcular las probabilidades de observación recibirá G junto con la observación en ese frame y los estados a calcular.
- $\Omega = \{\omega_i\}$: tabla que, para cada estado s_i, indica la cadena de caracteres que debe sacar en el proceso de *backtracking* o recuperación del camino correspondiente con la hipótesis final. Debido a la organización del vocabulario en forma de árbol para acelerar la búsqueda, el estado final de cada palabra es el que lleva asociada la cadena correspondiente a ésta.

Por otra parte, se define el *beam* como un parámetro del algoritmo. Éste indica el ratio entre dos verosimilitudes a partir del cual puede considerarse una despreciable frente a la otra.

C.2 ALGORITMO

C.2.1 *Inicialización*

El algoritmo comienza definiendo la matriz Φ donde se guardarán los caminos a medida que vayan recibiéndose observaciones, tras lo cual inicializa el vector de tokens con las probabilidades iniciales Π. Éste es un vector de N elementos en los cuales se guarda la probabilidad acumulada, normalizada al máximo, del token en ese estado. En el caso de que no haya un token activo en ese estado se guarda $-\infty$. La verosimilitud de los estados iniciales se pondera con las probabilidades de observación del primer vector de entrada $\mathbf{X}_0$.

```
1 Φ = zeros (N, T);
2 tok = Π;                              /* Inicialización de los tokens */
3 active = find (tok > −∞);
4 b = eval_st (X0, active, G);                  /* Cálculo de p. obs. */
5 tok(active) = tok(active) + b;
6 B = repmat (tok, 1, N);
```

Algoritmo 5: Inicialización

La función `eval_st` toma como argumentos las observaciones de ese instante $\mathbf{X}_0$, el modelo probabilístico G y los índices de los tokens activos, y devuelve las probabilidades de observación para sus estados. La última línea expande el vector de tokens en una matriz, necesaria para el paso de propagación. Cada fila contiene el mismo elemento en todas sus columnas: la probabilidad acumulada del token en ese estado o, en su defecto, $-\infty$ (ver Figura 35).

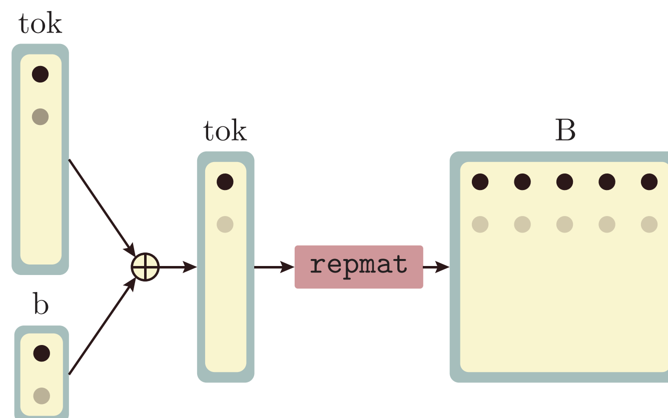


Figura 35: Líneas 11 y 12 en el Algoritmo 5

C.2.2 *Bucle de reconocimiento*

La siguiente parte del reconocedor se repite en bucle mientras hay nuevas observaciones de entrada. En una aplicación real, el bucle de reconocimiento es una función llamada desde otro programa, pero aquí se asume que el número de observaciones T es conocida de antemano.

```
1  for t = 1 : T − 1 do
2      B = B + A;                                        /* Paso de propagación */
3      [tok, i_max] = max (B, [], 1)';                   /* Purga */
4      max_tok = max (tok);
5      tok(tok < max_tok - beam) = −∞;                   /* Beam Search */
6      tok = tok - max_tok;                              /* Normalización */
7      active = find (tok > −∞);
8      Φ(active, t − 1) = i_max(active);                 /* Actualización de Φ */
9      b = eval_st (X_t, active, G);                     /* Cálculo de p. obs. */
10     tok(active) = tok(active) + b;                    /* Actualización de tok */
11     B = repmat (tok, 1, N);
```

Algoritmo 6: Bucle de reconocimiento

Tras el paso de propagación, la matriz B tiene, en cada elemento (i, j) distinto de $-\infty$, la probabilidad acumulada del token que se ha propagado desde el estado s_i hasta el s_j (Figura 36).

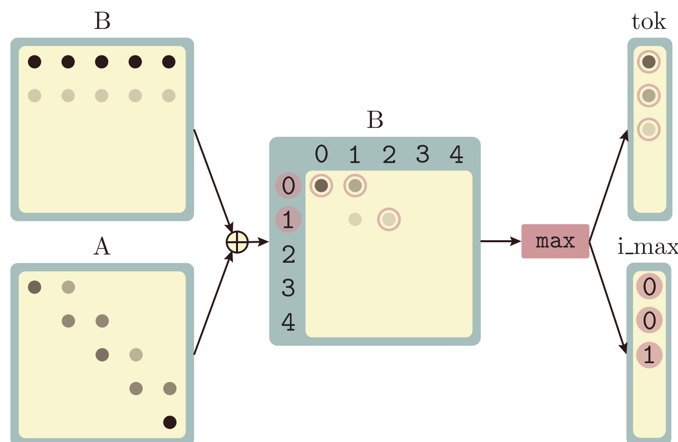


Figura 36: Líneas 2 y 3 en el Algoritmo 6

Posteriormente, se efectúa el paso de purga tomando el token con máxima probabilidad acumulada que ha llegado a un estado. La misma operación `max`, que recoge los máximos de cada columna de B en el vector tok, guarda en otro vector la filas o estados de los que provienen. Tras este paso se eliminan las hipótesis despreciables y

se normalizan las probabilidades acumuladas. En la columna $t-1$ de la matriz Φ se guardan los estados de procedencia de los tokens todavía activos, en las filas correspondientes a sus estados actuales. Después, se vuelven a ponderar las hipótesis por las probabilidades de observación y se prepara B para la siguiente iteración.

C.2.3 *Backtracking*

Finalmente, cuando no quedan más datos de entrada, se recupera la secuencia asociada al token con mayor probabilidad, almacenada en la matriz Φ. Tras recorrer ésta hacia atrás, la secuencia de estados queda guardada en un vector, st. El siguiente bucle comprueba la tabla Ω para cada estado de salida, imprimiendo por pantalla aquéllos que tienen una palabra asociada.

```
1  [~, i_best] = max (tok);              /* i_best: estado final más probable */
2  seq = zeros (1,T);
   /* Búsqueda de la secuencia de estados                                    */
3  st = i_best;
4  for t = T − 1 : −1 : 1 do
5      seq(t) = st;
6      st = Φ(st, t);
7  seq(0) = st;
   /* Mostrar la secuencia de palabras asociada                              */
8  q0 = −1;
9  for t = 1 : T − 1 do
10     q = seq(t);
       /* Sacar sólo una palabra si el estado ocupa varios frames            */
11     if q0 ≠ q then
12         ω = Ω(q);
13         if length (ω) > 0 then print (ω);
14     q0 = q;
```

Algoritmo 7: Backtracking

El algoritmo mostrado, representado en la Figura 37, define una matriz Φ de tamaño $N \times T$, pero para ahorrar espacio en memoria generalmente se limitan el número máximo de tokens y el de frames, quedando $nMax \times tMax$. Esta aproximación implica acudir al algoritmo de backtracking cada vez que el buffer se llena, recuperando la mejor hipótesis en ese momento sin la garantía de que lo vaya a ser en un futuro, y siguiendo el bucle de reconocimiento con el último estado de la secuencia como único token activo. También puede liberarse espacio de Φ sin eliminar hipótesis válidas haciendo el backtracking cuando todos los tokens se han propagado desde el mismo estado, ya que todas las secuencias compartirán hasta ese instante el mismo

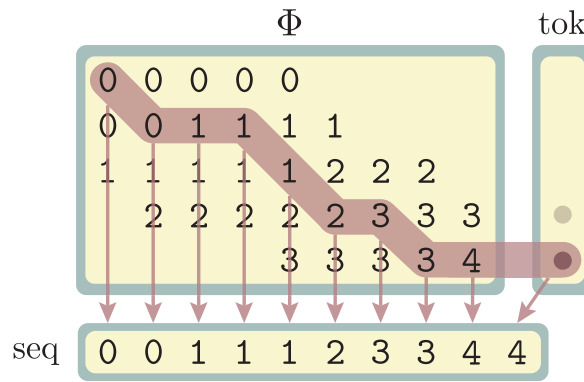


Figura 37: Líneas 1-7 en el Algoritmo 7

camino. En cualquiera de los dos casos esta gestión se hace al final del bucle de reconocimiento, antes de pasar a la siguiente iteración.

D

ESTRUCTURAS DE DATOS

Este apéndice detalla las estructuras de datos creadas para la implementación en CUDA C del algoritmo de reconocimiento.

Debido a que al comprimir una matriz se aumenta la dependencia entre sus elementos no nulos, lo cual dificulta la gestión en paralelo, se han definido distintos tipos de estructuras sparse dependiendo de su función en el algoritmo.

- `FSparse`, `ISparse`: Matrices sparse de tipo `float` e `int`, respectivamente. El vector `epr` indica el número de elementos no nulos en cada una de las `r` filas de la matriz. En cada fila de la matriz `val` se encuentran los valores no nulos de la matriz, apilados al principio de cada fila. En las mismas posiciones, los elementos de `colInd` indican la columna de cada uno de los valores. El valor `eprMax` determina el número máximo de elementos por fila.

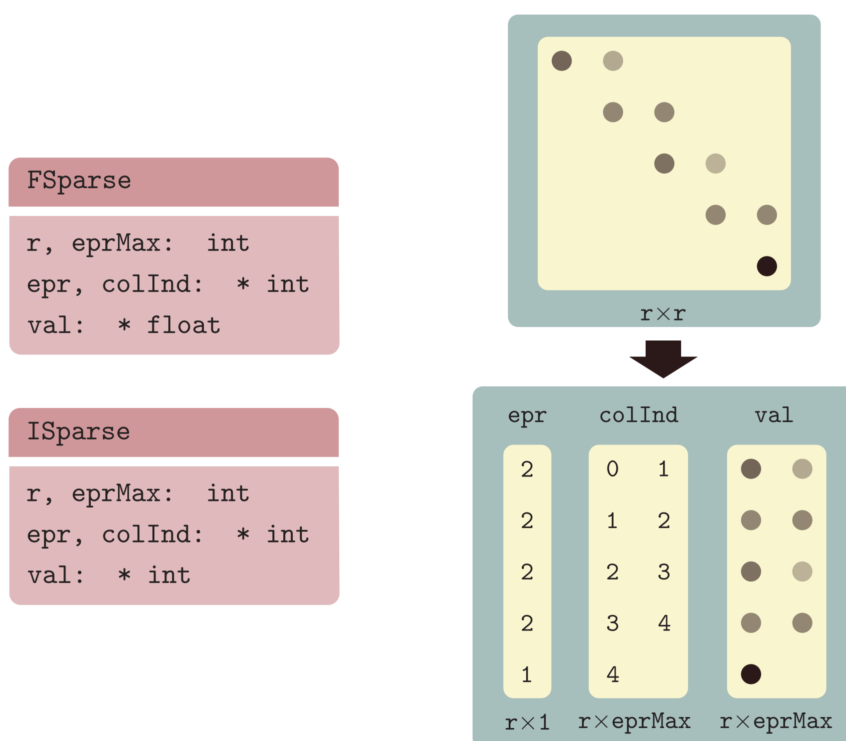


Figura 38: Estructuras `FSparse` e `ISparse`

- `VSparse`: Contiene el vector `val`, con `len` elementos de tipo `float`, `nMax` de ellos no nulos como máximo. No lleva ningún tipo de

compresión, por lo que no se trata de un vector sparse propiamente dicho, pero lleva asociado un vector de punteros `i` con las posiciones de los elementos no nulos, cuyo número se guarda en `n`, lo cual facilita la gestión de tokens. Opcionalmente puede declararse un vector de enteros `iPrev`, con un entero asociado a cada valor no nulo y en su misma posición, que servirá en el algoritmo para guardar el estado de procedencia de cada token.

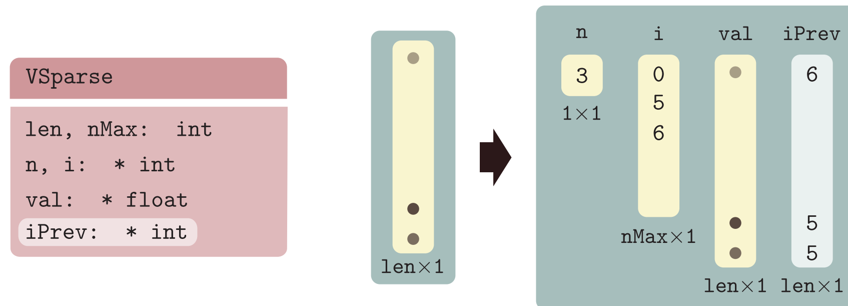


Figura 39: Estructura `VSparse`

- `Trans`: Matriz sparse de `float` que contiene las probabilidades de transición en escala logarítmica. Ésta es la estructura con la que más espacio se ahorra con una gestión de memoria eficiente, al ser una matriz de `r`×`r` elementos, con `r` el número de estados, generalmente muy elevado. El hecho de que su contenido no se modifique durante todo el algoritmo permite guardarla en un formato más comprimido. Los vectores `val` y `colInd` contienen los `n` valores no nulos de la matriz y sus respectivas columnas. El vector de enteros `iRow` guarda la posición inicial de los valores de cada fila en esos vectores, y en su último elemento contiene el número de elementos no nulos.

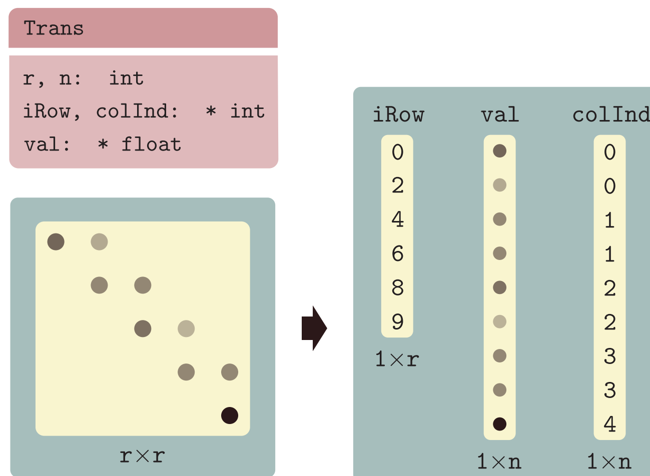


Figura 40: Estructura `Trans`

Se han implementado también funciones para poder definir estas estructuras en el host o en la GPU a partir de los parámetros de diseño, para inicializarlas a partir de un fichero, para imprimir su contenido y para copiarlo entre host y device y viceversa. A lo largo del código, se utiliza la convención de incluir `h_` y `d_` al principio de los nombres de las variables puntero para indicar si apuntan a posiciones de memoria en el host o en la GPU, respectivamente.

Otras variables se han agrupado en las siguientes estructuras de datos para simplificar el algoritmo principal, el cual llama a funciones que hacen uso de éstas, haciendo el código más legible y estructurado.

- `Token`: Esta estructura representa el vector de tokens, con las estructuras y variables que necesita para realizar el paso de propagación. El vector `tok`, de `maxTok` estados activos como máximo, guarda la información de los tokens activos en cada frame, mientras que `B` es la matriz donde se guardan los resultados intermedios de la propagación. En `maxVal` se guarda la probabilidad del token de mayor peso, obtenida mediante reducción con los vectores auxiliares `vMaxB` y `vAuxMaxB`. Finalmente, `eq` se usa para detectar si todos los tokens en un frame provienen del mismo estado, en cuyo caso pueden sacarse resultados parciales.

```
Token
h_tok, d_tok:  * VSparse
d_B: * FSparse
d_vMax, d_MaxAxB, d_maxVal:  * float
d_maxEprB: * int
d_eq:  * bool
h_eq:  bool
```

Figura 41: Estructura `Token`

- `Seq`: Secuencia de estados más probable, la cual se guarda en el vector `seq`, de `maxT` elementos como máximo. Los vectores de enteros `iFin` e `iFinAux` se emplean para buscar el estado final más probable por reducción, mientras que el entero `stFin`, inicializado a -1, guarda entre distintas llamadas a la función de backtracking el último estado recuperado. Esto es debido a que si un estado dura varios frames, en el caso de que lleve asociada una palabra solamente hay que mostrarla una vez por pantalla.
- `Phi`: Buffer circular para guardar los estados de procedencia de los tokens en cada frame temporal, para hacer posteriormente el backtracking. Estos estados se guardan en la estructura `ISparse`

P, de dimensiones maxT×maxTok. La fila de cada elemento en esta matriz indica el frame temporal, la columna indica el estado en el que se encuentra el token en ese momento y el valor, el estado desde el que se ha propagado. Las variables tIni y tFin, inicializadas a 0, son punteros a la posición inicial y final del buffer en un determinado momento. Con cada nuevo dato de entrada, tFin se incrementa (si llega al tamaño máximo de P, se pone a cero). Cuando se sacan resultados parciales, se actualiza tIni hasta la posición del último frame reconocido. Si tFin alcanza a tIni quiere decir que el buffer se ha llenado, en cuyo caso se llama a la función de backtracking, que busca la secuencia más probable hasta ese momento, tras lo cual puede vaciarse el buffer poniendo ambas variables a cero de nuevo.

```
Phi
nMax, maxT, tIni, tFin:  int
h_P, d_P: * ISparse
d_vMax, d_MaxAxB, d_maxVal:  * float
```

Figura 42: Estructura Phi

- Gauss: Parámetros estadísticos de los distintos estados, formados por n mezclas de Gaussianas apiladas en una matriz de tot×cols elementos (Gaussianas en total por parámetros de cada una). las tablas q2s e ini indican la correspondencia entre cada estado y su mezcla de Gaussianas (algunos estados comparten GMM) y la Gaussiana inicial de cada mezcla, mientras que fetch guarda durante los cálculos la posición de la probabilidad de observación de cada token dentro del vector pExp. En x se almacenan las distintas observaciones y gMask es una máscara que indica qué Gaussianas van a emplearse en ese frame. Los resultados finales se guardan en pObs.

```
Gauss
n, tot, cols:  int
h_q2s, d_q2s, h_ini, d_ini, d_fetch:  * int
h_Gauss, d_Gauss, d_x, d_pExp, d_pObs:  * float
d_vMax, d_vMaxAux, d_max:  * float
d_gMask:  * bool
```

Figura 43: Estructura Gauss

BIBLIOGRAFÍA

[1] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the *EM* algorithm. *Journal of the Royal Statistical Society*, 39(1):1–21, 1977.

[2] P. Dixon and S. Furui. Introduction to the use of WFSTs in speech and language processing. In *APSIPA Conference*, 2009.

[3] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[4] Stephan Kanthak, Hermann Ney, Michael Riley, and Mehryar Mohri. A comparison of two LVR search optimization techniques. In *INTERSPEECH*, 2002.

[5] L. R. Rabiner. *A Tutorial on HMM and selected Applications in Speech Recognition*, chapter 6.1, pages 267–295. Morgan Kaufmann, 1988.

[6] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. Htkbook (v3.3). Technical report, Cambridge University Engineering Department, 2005.